

Functional Data Structures

Tobias Nipkow

December 17, 2025

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	4
2	Creating Almost Complete Trees	13
3	Three-Way Comparison	19
4	Lists Sorted wrt $<$	20
5	List Insertion and Deletion	21
6	Specifications of Set ADT	24
7	Unbalanced Tree Implementation of Set	26
8	Association List Update and Deletion	29
9	Specifications of Map ADT	33
10	Unbalanced Tree Implementation of Map	34
11	Tree Rotations	36
12	Augmented Tree (Tree2)	39
13	Function <i>isin</i> for Tree2	40
14	Interval Trees	41

15 AVL Tree Implementation of Sets	48
16 Function <i>lookup</i> for Tree2	58
17 AVL Tree Implementation of Maps	58
18 AVL Tree with Balance Factors (1)	63
19 AVL Tree with Balance Factors (2)	67
20 Height-Balanced Trees	72
21 Red-Black Trees	80
22 Red-Black Tree Implementation of Sets	82
23 Alternative Deletion in Red-Black Trees	89
24 Red-Black Tree Implementation of Maps	93
25 2-3 Trees	96
26 2-3 Tree Implementation of Sets	97
27 2-3 Tree Implementation of Maps	106
28 2-3 Tree from List	109
29 2-3-4 Trees	112
30 2-3-4 Tree Implementation of Sets	114
31 2-3-4 Tree Implementation of Maps	126
32 1-2 Brother Tree Implementation of Sets	131
33 1-2 Brother Tree Implementation of Maps	143
34 AA Tree Implementation of Sets	148
35 AA Tree Implementation of Maps	160
36 Join-Based Implementation of Sets	165
37 Join-Based Implementation of Sets via RBTs	174
38 Braun Trees	181

39 Arrays via Braun Trees	185
40 Tries via Functions	202
41 Binary Tries and Patricia Tries	203
42 Ternary Tries	211
43 Queue Specification	213
44 Queue Implementation via 2 Lists	214
45 Priority Queue Specifications	216
46 Heaps	216
47 Leftist Heap	219
48 Binomial Priority Queue	227
49 The Median-of-Medians Selection Algorithm	242
50 Time Functions in Locales — An Example	268
51 Bibliographic Notes	271

1 Sorting

```
theory Sorting
  imports
    Complex_Main
    HOL-Library.Multiset
    HOL-Library.Time_Commands
begin
```

```
hide_const List.insort
```

```
declare Let_def [simp]
```

1.1 Insertion Sort

```
fun insort1 :: 'a::linorder  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  insort1 x [] = [x] |
  insort1 x (y#ys) =
    (if  $x \leq y$  then  $x\#y\#ys$  else  $y\#(\text{insort1 } x \text{ } ys)$ )
```

```
fun insort :: 'a::linorder list  $\Rightarrow$  'a list where
  insort [] = [] |
  insort (x#xs) = insort1 x (insort xs)
```

1.1.1 Functional Correctness

```
lemma mset_insort1:  $mset (\text{insort1 } x \text{ } xs) = \{x\} + mset \text{ } xs$ 
  by (induction xs) auto
```

```
lemma mset_insort:  $mset (\text{insort } xs) = mset \text{ } xs$ 
  by (induction xs) (auto simp: mset_insort1)
```

```
lemma set_insort1:  $set (\text{insort1 } x \text{ } xs) = \{x\} \cup set \text{ } xs$ 
  by (simp add: mset_insort1 flip: set_mset_mset)
```

```
lemma sorted_insort1:  $sorted (\text{insort1 } a \text{ } xs) = sorted \text{ } xs$ 
  by (induction xs) (auto simp: set_insort1)
```

```
lemma sorted_insort:  $sorted (\text{insort } xs)$ 
  by (induction xs) (auto simp: sorted_insort1)
```

1.1.2 Time Complexity

```
time_fun insort1
time_fun insort
```

```

lemma T_insort1_length:  $T\_insort1\ x\ xs \leq length\ xs + 1$ 
  by (induction xs) auto

lemma length_insort1:  $length\ (insort1\ x\ xs) = length\ xs + 1$ 
  by (induction xs) auto

lemma length_insort:  $length\ (insort\ xs) = length\ xs$ 
  by (metis Sorting.mset_insort size_mset)

lemma T_insort_length:  $T\_insort\ xs \leq (length\ xs + 1) ^ 2$ 
proof(induction xs)
  case Nil show ?case by simp
next
  case (Cons x xs)
  have  $T\_insort\ (x\#\ xs) = T\_insort\ xs + T\_insort1\ x\ (insort\ xs) + 1$  by
simp
  also have  $\dots \leq (length\ xs + 1) ^ 2 + T\_insort1\ x\ (insort\ xs) + 1$ 
    using Cons.IH by simp
  also have  $\dots \leq (length\ xs + 1) ^ 2 + length\ xs + 1 + 1$ 
    using T_insort1_length[of x insort xs] by (simp add: length_insort)
  also have  $\dots \leq (length(x\#\ xs) + 1) ^ 2$ 
    by (simp add: power2_eq_square)
  finally show ?case .
qed

```

1.2 Merge Sort

```

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

```

```

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

```

```

declare msort.simps [simp del]

```

1.2.1 Functional Correctness

```

lemma mset_merge:  $mset(merge\ xs\ ys) = mset\ xs + mset\ ys$ 

```

```

by(induction xs ys rule: merge.induct) auto

lemma mset_msort: mset (msort xs) = mset xs
proof(induction xs rule: msort.induct)
  case (1 xs)
  let ?n = length xs
  let ?ys = take (?n div 2) xs
  let ?zs = drop (?n div 2) xs
  show ?case
  proof cases
    assume ?n ≤ 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  $\Rightarrow$  'a list  $\Rightarrow$  nat where
  C_merge [] ys = 0 |
  C_merge xs [] = 0 |
  C_merge (x#xs) (y#ys) = 1 + (if x  $\leq$  y then C_merge xs (y#ys) else
  C_merge (x#xs) ys)

```

```

lemma C_merge_ub: C_merge xs ys  $\leq$  length xs + length ys
by (induction xs ys rule: C_merge.induct) auto

```

```

fun C_msort :: 'a::linorder list  $\Rightarrow$  nat where
  C_msort xs =
    (let n = length xs;
      ys = take (n div 2) xs;
      zs = drop (n div 2) xs
    in if n  $\leq$  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
by (induction xs ys rule: merge.induct) auto

```

```

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 = 2k  $\implies$  C_msort xs  $\leq$  k * 2k

```

```

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

```

```

lemma C_msort_log: length xs = 2k ⇒ C_msort xs ≤ length xs * log
  2 (length xs)
  using C_msort_le[of xs k]
  by (metis log2_of_power_eq mult.commute of_nat_mono 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 \Rightarrow 'a list **where**
merge_all [] = [] |
merge_all [xs] = xs |
merge_all xss = *merge_all* (*merge_adj* xss)

definition *msort_bu* :: ('a::linorder) list \Rightarrow 'a list **where**
msort_bu xs = *merge_all* (*map* ($\lambda x. [x]$) xs)

1.3.1 Functional Correctness

abbreviation *mset_mset* :: 'a list list \Rightarrow 'a multiset **where**
mset_mset xss $\equiv \sum \#$ (*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*:
 $\forall xs \in \text{set } xss. \text{sorted } xs \implies \forall xs \in \text{set } (\text{merge_adj } xss). \text{sorted } xs$
by(*induction xss rule: merge_adj.induct*) (*auto simp: sorted_merge*)

lemma *sorted_merge_all*:
 $\forall xs \in \text{set } xss. \text{sorted } xs \implies \text{sorted } (\text{merge_all } xss)$
by (*induction xss rule: merge_all.induct*) (*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 ( $\lambda x.$  [x]) xs)

```

```

lemma length_merge_adj:
  [ $\llbracket$  even(length xss);  $\forall xs \in \text{set } xss. \text{length } xs = m \rrbracket$ 
 $\implies \forall xs \in \text{set } (\text{merge\_adj } xss). \text{length } xs = 2*m$ 
by(induction xss rule: merge_adj.induct) (auto simp: length_merge)

```

```

lemma C_merge_adj:  $\forall xs \in \text{set } xss. \text{length } xs = m \implies C\_merge\_adj \ xss$ 
 $\leq m * \text{length } xss$ 
proof(induction xss rule: C_merge_adj.induct)
  case 1 thus ?case by simp
next
  case 2 thus ?case by simp
next
  case ( $\exists x \ y$ ) thus ?case using C_merge_ub[of x y] by (simp add: algebra_simps)
qed

```

```

lemma C_merge_all: [ $\llbracket \forall xs \in \text{set } xss. \text{length } xs = m; \text{length } xss = 2^k \rrbracket$ 
 $\implies C\_merge\_all \ xss \leq m * k * 2^k$ 
proof (induction xss arbitrary: k m rule: C_merge_all.induct)
  case 1 thus ?case by simp
next
  case 2 thus ?case by simp
next
  case ( $\exists xs \ ys \ xss$ )
    let ?xss = xs # ys # xss
    let ?xss2 = merge_adj ?xss
    obtain k' where k': k = Suc k' using 3.prem1(2)
    by (metis length_Cons nat.inject nat_power_eq_Suc_0_iff nat.exhaust)
    have even (length ?xss) using 3.prem1(2) k' by auto
    from length_merge_adj[OF this 3.prem1(1)]
    have *:  $\forall x \in \text{set}(\text{merge\_adj } ?xss). \text{length } x = 2*m$  .
    have **: length ?xss2 =  $2^{k'}$  using 3.prem1(2) k' by auto
    have C_merge_all ?xss = C_merge_adj ?xss + C_merge_all ?xss2 by
    simp
    also have ...  $\leq m * 2^{k'} + C\_merge\_all \ ?xss2$ 
    using 3.prem1(2) C_merge_adj[OF 3.prem1(1)] by (auto simp: algebra_simps)

```

```

bra_simps)
  also have ...  $\leq m * 2^k + (2*m) * k' * 2^{k'}$ 
    using 3.IH[OF **] by simp
  also have ...  $= m * k * 2^k$ 
    using k' by (simp add: algebra_simps)
  finally show ?case .
qed

```

corollary C_msort_bu : $length\ xs = 2^k \implies C_msort_bu\ xs \leq k * 2^k$
 using $C_merge_all[of\ map\ (\lambda x. [x])\ xs\ 1]$ by (simp add: $C_msort_bu_def$)

1.4 Quicksort

```

fun quicksort :: ('a::linorder) list  $\Rightarrow$  'a list where
  quicksort [] = [] |
  quicksort (x#xs) = quicksort (filter ( $\lambda y. y < x$ ) xs) @ [x] @ quicksort
    (filter ( $\lambda y. x \leq y$ ) xs)

```

lemma $mset_quicksort$: $mset\ (quicksort\ xs) = mset\ xs$
 by (induction xs rule: quicksort.induct) (auto simp: not_le)

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

lemma $sorted_quicksort$: $sorted\ (quicksort\ xs)$
proof (induction xs rule: quicksort.induct)
qed (auto simp: sorted_append set_quicksort)

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

hide__const $List.insort_key$

```

fun insort1_key :: ('a  $\Rightarrow$  'k::linorder)  $\Rightarrow$  'a  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  insort1_key f x [] = [x] |
  insort1_key f x (y # ys) = (if f x  $\leq$  f y then x # y # ys else y #
insort1_key f x ys)

```

```

fun insort_key :: ('a  $\Rightarrow$  'k::linorder)  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  insort_key f [] = [] |
  insort_key f (x # xs) = insort1_key f x (insort_key f xs)

```

1.5.1 Standard functional correctness

lemma $mset_insort1_key$: $mset\ (insort1_key\ f\ x\ xs) = \{\#x\#\} + mset\ xs$

by(*induction xs*) *simp_all*

lemma *mset_insort_key*: *mset (insort_key f xs) = mset xs*
by(*induction xs*) (*simp_all add: mset_insort1_key*)

lemma *set_insort1_key*: *set (insort1_key f x xs) = {x} ∪ set xs*
by (*induction xs*) *auto*

lemma *sorted_insort1_key*: *sorted (map f (insort1_key f a xs)) = sorted (map f xs)*
by(*induction xs*)(*auto simp: set_insort1_key*)

lemma *sorted_insort_key*: *sorted (map f (insort_key f xs))*
by(*induction xs*)(*simp_all add: sorted_insort1_key*)

1.5.2 Stability

lemma *insort1_is_Cons*: $\forall x \in \text{set } xs. f a \leq f x \implies \text{insort1_key } f a xs = a \# xs$
by (*cases xs*) *auto*

lemma *filter_insort1_key_neg*:
 $\neg P x \implies \text{filter } P (\text{insort1_key } f x xs) = \text{filter } P xs$
by (*induction xs*) *simp_all*

lemma *filter_insort1_key_pos*:
 $\text{sorted } (\text{map } f xs) \implies P x \implies \text{filter } P (\text{insort1_key } f x xs) = \text{insort1_key } f x (\text{filter } P xs)$
by (*induction xs*) (*auto, subst insort1_is_Cons, auto*)

lemma *sort_key_stable*: $\text{filter } (\lambda y. f y = k) (\text{insort_key } f xs) = \text{filter } (\lambda y. f y = k) xs$

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_insort1_key_neg*)
next
case *True*
have $\text{filter } (\lambda y. f y = k) (\text{insort_key } f (a \# xs))$
 $= \text{filter } (\lambda y. f y = k) (\text{insort1_key } f a (\text{insort_key } f xs))$ **by** *simp*

```

    also have ... = insert1_key f a (filter (λy. f y = k) (insert_key f xs))
      by (simp add: True filter_insert1_key_pos sorted_insert_key)
    also have ... = insert1_key f a (filter (λy. f y = k) xs) by (simp add:
Cons.IH)
    also have ... = a # (filter (λy. f y = k) xs) by (simp add: True
insert1_is_Cons)
    also have ... = filter (λy. f y = k) (a # xs) by (simp add: True)
    finally show ?thesis .
qed
qed

```

1.6 Uniqueness of Sorting

```

lemma sorting_unique:
  assumes mset ys = mset xs sorted xs sorted ys
  shows xs = ys
  using assms
proof (induction xs arbitrary: ys)
  case (Cons x xs ys')
  obtain y ys where ys': ys' = y # ys
    using Cons.prem by (cases ys') auto
  have x = y
    using Cons.prem unfolding ys'
  proof (induction x y arbitrary: xs ys rule: linorder_wlog)
    case (le x y xs ys)
    have x ∈# mset (x # xs)
      by simp
    also have mset (x # xs) = mset (y # ys)
      using le by simp
    finally show x = y
      using le by auto
  qed (simp_all add: eq_commute)
  thus ?case
    using Cons.prem Cons.IH[of ys] by (auto simp: ys')
qed auto
end

```

2 Creating Almost Complete Trees

```

theory Balance
imports
  HOL-Library.Tree_Real

```

begin

fun *bal* :: *nat* \Rightarrow '*a list* \Rightarrow '*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* \Rightarrow '*a list* \Rightarrow '*a tree* **where**
bal_list *n xs* = *fst* (*bal* *n xs*)

definition *balance_list* :: '*a list* \Rightarrow '*a tree* **where**
balance_list *xs* = *bal_list* (*length* *xs*) *xs*

definition *bal_tree* :: *nat* \Rightarrow '*a tree* \Rightarrow '*a tree* **where**
bal_tree *n t* = *bal_list* *n* (*inorder* *t*)

definition *balance_tree* :: '*a tree* \Rightarrow '*a tree* **where**
balance_tree *t* = *bal_tree* (*size* *t*) *t*

lemma *bal_simps*:

bal 0 *xs* = (*Leaf*, *xs*)
n > 0 \implies
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*:

$\llbracket n \leq \text{length } xs; \text{bal } n \text{ } xs = (t, zs) \rrbracket$
 $\implies xs = \text{inorder } t @ zs \wedge \text{size } t = n$

proof(*induction* *n* *arbitrary: xs t zs* *rule: less_induct*)

case (*less* *n*) **show** ?*case*

proof *cases*

assume *n* = 0 **thus** ?*thesis* **using** *less.prem*s **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 ≤ 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 ≤ 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 $n \leq \text{length } xs$ (or $n \leq \text{size } t$) that they come with. They could take advantage of the fact that $\text{bal } xs \ n$ yields a result even if $\text{length } xs < n$. In that case the result will contain one or more occurrences of $\text{hd } []$. However, this is counter-intuitive and does not reflect the execution in an eager functional language.

```

lemma bal_length: [ n ≤ length xs; bal n xs = (t,zs) ] ⇒ length zs =
length xs - n
using bal_inorder by fastforce

```

```

corollary size_bal_list[simp]: n ≤ 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]: $n \leq \text{size } t \implies \text{size}(\text{bal_tree } n \ t) = n$
by(*simp add: bal_tree_def*)

corollary *size_balance_tree*[simp]: $\text{size}(\text{balance_tree } t) = \text{size } t$
by(*simp add: balance_tree_def*)

lemma *min_height_bal*:

$\llbracket n \leq \text{length } xs; \text{bal } n \ xs = (t, zs) \rrbracket \implies \text{min_height } t = \text{nat}(\lfloor \log 2 \ (n + 1) \rfloor)$

proof(*induction n arbitrary: xs t zs rule: less_induct*)

case (*less n*)

show *?case*

proof *cases*

assume $n = 0$ **thus** *?thesis* **using** *less.prem1* **by** (*simp add: bal_simps*)

next

assume [*arith*]: $n \neq 0$

let $?m = n \text{ div } 2$ **let** $?m' = n - 1 - ?m$

from *less.prem1* **obtain** $l \ r \ ys$ **where**

$b1: \text{bal } ?m \ xs = (l, ys)$ **and**

$b2: \text{bal } ?m' \ (tl \ ys) = (r, zs)$ **and**

$t: t = \langle l, \text{hd } ys, r \rangle$

by(*auto simp: bal_simps split: prod.splits*)

let $?hl = \text{nat}(\text{floor}(\log 2 \ (?m + 1)))$

let $?hr = \text{nat}(\text{floor}(\log 2 \ (?m' + 1)))$

have *IH1*: $\text{min_height } l = ?hl$ **using** *less.IH*[*OF* __ *b1*] *less.prem1*

by *simp*

have *IH2*: $\text{min_height } r = ?hr$

using *less.prem1* *bal_length*[*OF* __ *b1*] *b2* **by**(*intro less.IH*) *auto*

have $(n+1) \text{ div } 2 \geq 1$ **by** *arith*

hence $0: \log 2 \ ((n+1) \text{ div } 2) \geq 0$ **by** *simp*

have $?m' \leq ?m$ **by** *arith*

hence *le*: $?hr \leq ?hl$ **by**(*simp add: nat_mono floor_mono*)

have $\text{min_height } t = \min ?hl ?hr + 1$ **by** (*simp add: t IH1 IH2*)

also have $\dots = ?hr + 1$ **using** *le* **by** (*simp add: min_absorb2*)

also have $?m' + 1 = (n+1) \text{ div } 2$ **by** *linarith*

also have $\text{nat}(\text{floor}(\log 2 \ ((n+1) \text{ div } 2))) + 1$

$= \text{nat}(\text{floor}(\log 2 \ ((n+1) \text{ div } 2) + 1))$

using 0 **by** *linarith*

also have $\dots = \text{nat}(\text{floor}(\log 2 \ (n + 1)))$

using *floor_log2_div2*[*of* $n+1$] **by** (*simp add: log_mult*)

finally show *?thesis* .

qed

qed

lemma *height_bal*:

$\llbracket n \leq \text{length } xs; \text{ bal } n \text{ } xs = (t, zs) \rrbracket \implies \text{height } t = \text{nat } \lceil \log 2 (n + 1) \rceil$

proof(*induction n arbitrary: xs t zs rule: less_induct*)

case (*less n*) **show** *?case*

proof *cases*

assume $n = 0$ **thus** *?thesis*

using *less.prem*s **by** (*simp add: bal_simps*)

next

assume [*arith*]: $n \neq 0$

let $?m = n \text{ div } 2$ **let** $?m' = n - 1 - ?m$

from *less.prem*s **obtain** $l \ r \ ys$ **where**

$b1$: $\text{bal } ?m \text{ } xs = (l, ys)$ **and**

$b2$: $\text{bal } ?m' \text{ } (tl \ ys) = (r, zs)$ **and**

t : $t = \langle l, \text{hd } ys, r \rangle$

by(*auto simp: bal_simps split: prod.splits*)

let $?hl = \text{nat } \lceil \log 2 (?m + 1) \rceil$

let $?hr = \text{nat } \lceil \log 2 (?m' + 1) \rceil$

have $IH1$: $\text{height } l = ?hl$ **using** *less.IH*[*OF* $__\ b1$] *less.prem*s(1) **by**

simp

have $IH2$: $\text{height } r = ?hr$

using $b2$ *bal_length*[*OF* $__\ b1$] *less.prem*s(1) **by**(*intro less.IH*) *auto*

have 0 : $\log 2 (?m + 1) \geq 0$ **by** *simp*

have $?m' \leq ?m$ **by** *arith*

hence le : $?hr \leq ?hl$

by(*simp add: nat_mono ceiling_mono del: nat_ceiling_le_eq*)

have $\text{height } t = \max ?hl ?hr + 1$ **by** (*simp add: t IH1 IH2*)

also have $\dots = ?hl + 1$ **using** le **by** (*simp add: max_absorb1*)

also have $\dots = \text{nat } \lceil \log 2 (?m + 1) + 1 \rceil$ **using** 0 **by** *linarith*

also have $\dots = \text{nat } \lceil \log 2 (n + 1) \rceil$

using *ceiling_log2_div2*[*of* $n+1$] **by** (*simp*)

finally show *?thesis* .

qed

qed

lemma *acomplete_bal*:

assumes $n \leq \text{length } xs$ $\text{bal } n \text{ } xs = (t, ys)$ **shows** *acomplete t*

unfolding *acomplete_def*

using *height_bal*[*OF* *assms*] *min_height_bal*[*OF* *assms*]

by *linarith*

lemma *height_bal_list*:

$n \leq \text{length } xs \implies \text{height } (\text{bal_list } n \text{ } xs) = \text{nat } \lceil \log 2 (n + 1) \rceil$

unfolding *bal_list_def* **by** (*metis height_bal prod.collapse*)

lemma *height_balance_list*:
 $\text{height } (\text{balance_list } xs) = \text{nat } \lceil \log 2 (\text{length } xs + 1) \rceil$
by (*simp add: balance_list_def height_bal_list*)

corollary *height_bal_tree*:
 $n \leq \text{size } t \implies \text{height } (\text{bal_tree } n \ t) = \text{nat } \lceil \log 2 (n + 1) \rceil$
unfolding *bal_list_def bal_tree_def*
by (*metis bal_list_def height_bal_list length_inorder*)

corollary *height_balance_tree*:
 $\text{height } (\text{balance_tree } t) = \text{nat } \lceil \log 2 (\text{size } t + 1) \rceil$
by (*simp add: bal_tree_def balance_tree_def height_bal_list*)

corollary *acomplete_bal_list[simp]*: $n \leq \text{length } xs \implies \text{acomplete } (\text{bal_list } n \ xs)$
unfolding *bal_list_def* **by** (*metis acomplete_bal prod.collapse*)

corollary *acomplete_balance_list[simp]*: $\text{acomplete } (\text{balance_list } xs)$
by (*simp add: balance_list_def*)

corollary *acomplete_bal_tree[simp]*: $n \leq \text{size } t \implies \text{acomplete } (\text{bal_tree } n \ t)$
by (*simp add: bal_tree_def*)

corollary *acomplete_balance_tree[simp]*: $\text{acomplete } (\text{balance_tree } t)$
by (*simp add: balance_tree_def*)

lemma *wbalanced_bal*: $\llbracket n \leq \text{length } xs; \text{bal } n \ xs = (t, ys) \rrbracket \implies \text{wbanced } t$
proof(*induction n arbitrary: xs t ys rule: less_induct*)
case (*less n*)
show *?case*
proof *cases*
assume $n = 0$
thus *?thesis* **using** *less.prem1* **by**(*simp add: bal_simps*)
next
assume [*arith*]: $n \neq 0$
with *less.prem1* **obtain** $l \ ys \ r \ zs$ **where**
 $\text{rec1: bal } (n \text{ div } 2) \ xs = (l, ys)$ **and**
 $\text{rec2: bal } (n - 1 - n \text{ div } 2) \ (tl \ ys) = (r, zs)$ **and**
 $t = \langle l, hd \ ys, r \rangle$
by(*auto simp add: bal_simps split: prod.splits*)
have l : *wbalanced* l **using** *less.IH[OF rec1]* *less.prem1* **by** *linarith*
have *wbalanced* r

```

    using rec1 rec2 bal_length[OF _ rec1] less.prem1 by(intro less.IH)
  auto
    with l t bal_length[OF _ rec1] less.prem1 bal_inorder[OF _ rec1]
    bal_inorder[OF _ rec2]
    show ?thesis by auto
qed
qed

```

An alternative proof via *wbalanced* $?t \implies \text{acomplete } ?t$:

```

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 ⟷ x = y
and GT[simp]: cmp x y = GT ⟷ x > y

```

by (*auto simp: cmp_def*)

lemma *case_cmp_if*[*simp*]: (*case c of EQ \Rightarrow e | LT \Rightarrow l | GT \Rightarrow 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 \Rightarrow bool* **where**
sorted \equiv *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_on_less, simp*]

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

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

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

lemma *sorted_snoc'*: *ASSUMPTION (sorted (xs @ [y])) \Longrightarrow 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  $\implies$ 
   $sorted(xs @ y \# ys) = (sorted(xs @ [y]) \wedge sorted(y \# ys))$ 
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]))  $\implies x \leq y \implies sorted\ (xs @ [y])$ 
by (auto simp add: sorted_wrt_append ASSUMPTION_def)

lemma sorted_Cons_le:
  ASSUMPTION(sorted(x # xs))  $\implies y \leq x \implies sorted\ (y \# xs)$ 
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  $\Rightarrow$  'a list  $\Rightarrow$  '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  $\cup$  {x}
by(induction xs) auto
```

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

```
lemma ins_list_sorted: sorted (xs @ [a])  $\implies$ 
  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, $\text{sorted } (?xs @ [?a]) \implies \text{ins_list } ?x (?xs @ ?a \# ?ys) =$ (if $?x < ?a$ then $\text{ins_list } ?x ?xs @ ?a \# ?ys$ else $?xs @ \text{ins_list } ?x (?a \# ?ys)$) suffices, but the following two corollaries speed up proofs.

```
corollary ins_list_sorted1: sorted (xs @ [a])  $\implies$  a  $\leq$  x  $\implies$ 
  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])  $\implies$  x < a  $\implies$ 
  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)  $\implies$  ins_list x xs = x # xs
by (induction xs) auto
```

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

5.3 Delete one occurrence of an element from a list:

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

```
lemma del_list_idem: x  $\notin$  set xs  $\implies$  del_list x xs = xs
```

by (*induct xs*) *simp_all*

lemma *set_del_list*:

$\text{sorted } xs \implies \text{set } (\text{del_list } x \text{ } xs) = \text{set } xs - \{x\}$

by(*induct xs*) (*auto simp: sorted_Cons_iff*)

lemma *sorted_del_list*: $\text{sorted } xs \implies \text{sorted}(\text{del_list } x \text{ } xs)$

apply(*induction xs rule: induct_list012*)

apply *auto*

by (*meson order.strict_trans sorted_Cons_iff*)

lemma *del_list_sorted*: $\text{sorted } (xs @ a \# ys) \implies$

$\text{del_list } x \text{ } (xs @ a \# ys) = (\text{if } x < a \text{ then } \text{del_list } x \text{ } xs @ a \# ys \text{ else } xs @ \text{del_list } x \text{ } (a \# ys))$

by(*induction xs*)

(*fastforce simp: sorted_lems sorted_Cons_iff intro!: del_list_idem*)+

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

corollary *del_list_sorted1*: $\text{sorted } (xs @ a \# ys) \implies a \leq x \implies$

$\text{del_list } x \text{ } (xs @ a \# ys) = xs @ \text{del_list } x \text{ } (a \# ys)$

by (*auto simp: del_list_sorted*)

corollary *del_list_sorted2*: $\text{sorted } (xs @ a \# ys) \implies x < a \implies$

$\text{del_list } x \text{ } (xs @ a \# ys) = \text{del_list } x \text{ } xs @ a \# ys$

by (*auto simp: del_list_sorted*)

corollary *del_list_sorted3*:

$\text{sorted } (xs @ a \# ys @ b \# zs) \implies x < b \implies$

$\text{del_list } x \text{ } (xs @ a \# ys @ b \# zs) = \text{del_list } x \text{ } (xs @ a \# ys) @ b \# zs$

by (*auto simp: del_list_sorted sorted_lems*)

corollary *del_list_sorted4*:

$\text{sorted } (xs @ a \# ys @ b \# zs @ c \# us) \implies x < c \implies$

$\text{del_list } x \text{ } (xs @ a \# ys @ b \# zs @ c \# us) = \text{del_list } x \text{ } (xs @ a \# ys @ b \# zs) @ c \# us$

by (*auto simp: del_list_sorted sorted_lems*)

corollary *del_list_sorted5*:

$\text{sorted } (xs @ a \# ys @ b \# zs @ c \# us @ d \# vs) \implies x < d \implies$

$\text{del_list } x \text{ } (xs @ a \# ys @ b \# zs @ c \# us @ d \# vs) =$

$\text{del_list } x \text{ } (xs @ a \# ys @ b \# zs @ c \# us) @ d \# vs$

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*) \implies *del_list* *x* *xs* = *xs*

by (*induction xs*) (*fastforce simp: sorted_Cons_iff*) +

lemma *del_list_sorted_app*:

sorted (*xs* @ [*x*]) \implies *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* \implies *isin s x* = (*x* \in *set s*)

assumes *set_insert*: *invar s* \implies *set* (*insert x s*) = *set s* \cup {*x*}

assumes *set_delete*: *invar s* \implies *set* (*delete x s*) = *set s* - {*x*}

assumes *invar_empty*: *invar empty*

assumes *invar_insert*: *invar s* \implies *invar* (*insert x s*)

assumes *invar_delete*: *invar s* \implies *invar* (*delete x s*)

lemmas (**in** *Set*) *set_specs* =

set_empty set_isin set_insert set_delete invar_empty invar_insert invar_delete

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


```

locale Set_by_Ordered =
fixes empty :: 't
fixes insert :: 'a::linorder  $\Rightarrow$  't  $\Rightarrow$  't
fixes delete :: 'a  $\Rightarrow$  't  $\Rightarrow$  't
fixes isin :: 't  $\Rightarrow$  'a  $\Rightarrow$  bool
fixes inorder :: 't  $\Rightarrow$  'a list
fixes inv :: 't  $\Rightarrow$  bool
assumes inorder_empty: inorder empty = []
assumes isin: inv t  $\wedge$  sorted(inorder t)  $\implies$ 
  isin t x = (x  $\in$  set (inorder t))
assumes inorder_insert: inv t  $\wedge$  sorted(inorder t)  $\implies$ 
  inorder(insert x t) = ins_list x (inorder t)
assumes inorder_delete: inv t  $\wedge$  sorted(inorder t)  $\implies$ 
  inorder(delete x t) = del_list x (inorder t)
assumes inorder_inv_empty: inv empty
assumes inorder_inv_insert: inv t  $\wedge$  sorted(inorder t)  $\implies$  inv(insert x t)
assumes inorder_inv_delete: inv t  $\wedge$  sorted(inorder t)  $\implies$  inv(delete x t)

```

begin

It implements the traditional specification:

```

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

```

```

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

```

sublocale Set

```

  empty insert delete isin set invar
proof(standard, goal_cases)
  case 1 show ?case by (auto simp: inorder_empty set_def)
next
  case 2 thus ?case by(simp add: isin invar_def set_def)
next
  case 3 thus ?case by(simp add: inorder_insert set_ins_list set_def 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  $\Rightarrow$  's  $\Rightarrow$  's +
  fixes union :: 's  $\Rightarrow$  's  $\Rightarrow$  's
  fixes inter :: 's  $\Rightarrow$  's  $\Rightarrow$  's
  fixes diff :: 's  $\Rightarrow$  's  $\Rightarrow$  's
  assumes set_union:  $\llbracket \text{invar } s1; \text{invar } s2 \rrbracket \Longrightarrow \text{set}(\text{union } s1 \ s2) = \text{set } s1 \cup \text{set } s2$ 
  assumes set_inter:  $\llbracket \text{invar } s1; \text{invar } s2 \rrbracket \Longrightarrow \text{set}(\text{inter } s1 \ s2) = \text{set } s1 \cap \text{set } s2$ 
  assumes set_diff:  $\llbracket \text{invar } s1; \text{invar } s2 \rrbracket \Longrightarrow \text{set}(\text{diff } s1 \ s2) = \text{set } s1 - \text{set } s2$ 
  assumes invar_union:  $\llbracket \text{invar } s1; \text{invar } s2 \rrbracket \Longrightarrow \text{invar}(\text{union } s1 \ s2)$ 
  assumes invar_inter:  $\llbracket \text{invar } s1; \text{invar } s2 \rrbracket \Longrightarrow \text{invar}(\text{inter } s1 \ s2)$ 
  assumes invar_diff:  $\llbracket \text{invar } s1; \text{invar } s2 \rrbracket \Longrightarrow \text{invar}(\text{diff } s1 \ s2)$ 

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  $\Rightarrow$  '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 |

```

$GT \Rightarrow isin\ r\ x)$

hide_const (**open**) *insert*

fun *insert* :: 'a::linorder \Rightarrow 'a tree \Rightarrow 'a tree **where**

insert *x* *Leaf* = *Node* *Leaf* *x* *Leaf* |

insert *x* (*Node* *l* *a* *r*) =

(*case cmp* *x* *a* of

LT \Rightarrow *Node* (*insert* *x* *l*) *a* *r* |

EQ \Rightarrow *Node* *l* *a* *r* |

GT \Rightarrow *Node* *l* *a* (*insert* *x* *r*))

Deletion by replacing:

fun *split_min* :: 'a tree \Rightarrow '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 \Rightarrow 'a tree \Rightarrow 'a tree **where**

delete *x* *Leaf* = *Leaf* |

delete *x* (*Node* *l* *a* *r*) =

(*case cmp* *x* *a* of

LT \Rightarrow *Node* (*delete* *x* *l*) *a* *r* |

GT \Rightarrow *Node* *l* *a* (*delete* *x* *r*) |

EQ \Rightarrow 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 \Rightarrow 'a tree \Rightarrow '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 \Rightarrow *Node* *t1* *a* (*Node* *Leaf* *b* *t4*) |

Node *u2* *x* *u3* \Rightarrow *Node* (*Node* *t1* *a* *u2*) *x* (*Node* *u3* *b* *t4*))

fun *delete2* :: 'a::linorder \Rightarrow 'a tree \Rightarrow 'a tree **where**

delete2 *x* *Leaf* = *Leaf* |

delete2 *x* (*Node* *l* *a* *r*) =

(*case cmp* *x* *a* of

LT \Rightarrow *Node* (*delete2* *x* *l*) *a* *r* |

GT \Rightarrow *Node* *l* *a* (*delete2* *x* *r*) |

EQ \Rightarrow *join* *l* *r*)

7.1 Functional Correctness Proofs

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

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

lemma *inorder_insert*:

sorted(inorder t) \implies 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') \implies t \neq Leaf \implies 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) \implies 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* = $\lambda_.$ *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) \implies 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* = $\lambda_.$ *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* \equiv *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  $\Rightarrow$  'a  $\Rightarrow$  'b option where
map_of [] = ( $\lambda x$ . None) |
map_of ((a,b)#ps) = ( $\lambda x$ . if x=a then Some b else map_of ps x)

```

Updating an association list:

```

fun upd_list :: 'a::linorder  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) list  $\Rightarrow$  ('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  $\Rightarrow$  ('a*'b)list  $\Rightarrow$  ('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 \Rightarrow map_of qs x | Some y \Rightarrow 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 \leq 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*: $\llbracket \text{sorted } (\text{map } \text{fst } ps \ @ \ [a]); x < a \rrbracket \implies$
 $\text{upd_list } x \ y \ (ps \ @ \ (a,b) \ # \ qs) = \text{upd_list } x \ y \ ps \ @ \ (a,b) \ # \ qs$
by (*auto simp: upd_list_sorted*)

corollary *upd_list_sorted2*: $\llbracket \text{sorted } (\text{map } \text{fst } ps \ @ \ [a]); a \leq x \rrbracket \implies$
 $\text{upd_list } x \ y \ (ps \ @ \ (a,b) \ # \ qs) = ps \ @ \ \text{upd_list } x \ y \ ((a,b) \ # \ qs)$
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*:
 $\text{sorted1 } ((x,y) \ # \ xs) \implies \text{upd_list } x \ y \ xs = (x,y) \ # \ xs$
by (*induction xs*) *auto*

lemma *upd_list_snoc*:
 $\text{sorted1 } (xs \ @ \ [(x,y)]) \implies \text{upd_list } x \ y \ xs = xs \ @ \ [(x,y)]$
by(*induction xs*) (*auto simp add: sorted_mid_iff2*)

8.3 Lemmas for *del_list*

lemma *sorted_del_list*: $\text{sorted1 } ps \implies \text{sorted1 } (\text{del_list } x \ ps)$
apply(*induction ps*)
apply *simp*
apply(*case_tac ps*)
apply (*auto simp: sorted_Cons_le*)
done

lemma *del_list_idem*: $x \notin \text{set}(\text{map } \text{fst } xs) \implies \text{del_list } x \ xs = xs$
by (*induct xs*) *auto*

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

In principle, $\text{sorted1 } (?ps \ @ \ (?a, ?b) \ # \ ?qs) \implies \text{del_list } ?x \ (?ps \ @ \ (?a, ?b) \ # \ ?qs) = (\text{if } ?x < ?a \text{ then } \text{del_list } ?x \ ?ps \ @ \ (?a, ?b) \ # \ ?qs \text{ else } ?ps \ @ \ \text{del_list } ?x \ ((?a, ?b) \ # \ ?qs))$ suffices, but the following corollaries speed up proofs.

corollary *del_list_sorted1*: $\text{sorted1 } (xs \ @ \ (a,b) \ # \ ys) \implies a \leq x \implies$
 $\text{del_list } x \ (xs \ @ \ (a,b) \ # \ ys) = xs \ @ \ \text{del_list } x \ ((a,b) \ # \ ys)$

by (*auto simp: del_list_sorted*)

lemma *del_list_sorted2*: *sorted1 (xs @ (a,b) # ys) \implies x < a \implies*
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) \implies x < b \implies
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) \implies x < c \implies
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) \implies
x < d \implies
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) \implies del_list x xs =*
xs
by(*induction xs*)(*fastforce simp: sorted_wrt_Cons*)+

lemma *del_list_sorted_app*:
sorted(map fst xs @ [x]) \implies 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 $'a \Rightarrow 'b$ option based specification:

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

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

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

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

begin
```

It implements the traditional specification:

```

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

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

end

end

```

10 Unbalanced Tree Implementation of Map

```

theory Tree_Map
imports
  Tree_Set
  Map_Specs
begin

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

```

```

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

```

```

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

```

```

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

```

10.1 Functional Correctness Proofs

```

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

```

```

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

```

```

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

```

```

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

```

qed *auto*

end

11 Tree Rotations

theory *Tree_Rotations*
imports *HOL-Library.Tree*
begin

How to transform a tree into a list and into any other tree (with the same *inorder*) by rotations.

fun *is_list* :: '*a* tree \Rightarrow bool **where**
is_list (Node *l* *r*) = (*l* = Leaf \wedge *is_list* *r*) |
is_list Leaf = True

Termination proof via measure function. NB *size t - rlen t* works for the actual rotation equation but not for the second equation.

fun *rlen* :: '*a* tree \Rightarrow nat **where**
rlen Leaf = 0 |
rlen (Node *l* *x* *r*) = *rlen* *r* + 1

lemma *rlen_le_size*: *rlen t* \leq *size t*
by(*induction t*) *auto*

11.1 Without positions

function (*sequential*) *list_of* :: '*a* tree \Rightarrow '*a* tree **where**
list_of (Node (Node *A* *a* *B*) *b* *C*) = *list_of* (Node *A* *a* (Node *B* *b* *C*)) |
list_of (Node Leaf *a* *A*) = Node Leaf *a* (*list_of* *A*) |
list_of Leaf = Leaf
by *pat_completeness auto*

termination

proof

let *?R* = *measure*($\lambda t. 2 * \text{size } t - \text{rlen } t$)
show *wf ?R* **by** (*auto simp add: mlex_prod_def*)

fix *A a B b C*

show (Node *A* *a* (Node *B* *b* *C*), Node (Node *A* *a* *B*) *b* *C*) \in *?R*
using *rlen_le_size[of C]* **by**(*simp*)

fix *a A* **show** (*A*, Node Leaf *a* *A*) \in *?R* **using** *rlen_le_size[of A]* **by**(*simp*)
qed

lemma *is_list_rot*: *is_list*(*list_of* *t*)
by (*induction* *t* *rule*: *list_of.induct*) *auto*

lemma *inorder_rot*: *inorder*(*list_of* *t*) = *inorder* *t*
by (*induction* *t* *rule*: *list_of.induct*) *auto*

11.2 With positions

datatype *dir* = *L* | *R*

type_synonym *pos* = *dir list*

function (*sequential*) *rotR_poss* :: '*a* tree \Rightarrow *pos list* **where**
rotR_poss (*Node* (*Node* *A* *a* *B*) *b* *C*) = [] # *rotR_poss* (*Node* *A* *a* (*Node* *B* *b* *C*)) |
rotR_poss (*Node* *Leaf* *a* *A*) = *map* (*Cons* *R*) (*rotR_poss* *A*) |
rotR_poss *Leaf* = []
by *pat_completeness auto*

termination

proof

let *?R* = *measure*($\lambda t. 2 * \text{size } t - \text{rlen } t$)
show *wf* *?R* **by** (*auto simp add: mlex_prod_def*)

fix *A a B b C*

show (*Node* *A* *a* (*Node* *B* *b* *C*), *Node* (*Node* *A* *a* *B*) *b* *C*) \in *?R*
using *rlen_le_size[of C]* **by**(*simp*)

fix *a A* **show** (*A*, *Node* *Leaf* *a* *A*) \in *?R* **using** *rlen_le_size[of A]* **by**(*simp*)
qed

fun *rotR* :: '*a* tree \Rightarrow '*a* tree **where**
rotR (*Node* (*Node* *A* *a* *B*) *b* *C*) = *Node* *A* *a* (*Node* *B* *b* *C*)

fun *rotL* :: '*a* tree \Rightarrow '*a* tree **where**
rotL (*Node* *A* *a* (*Node* *B* *b* *C*)) = *Node* (*Node* *A* *a* *B*) *b* *C*

fun *apply_at* :: ('*a* tree \Rightarrow '*a* tree) \Rightarrow *pos* \Rightarrow '*a* tree \Rightarrow '*a* tree **where**
apply_at *f* [] *t* = *f* *t*
| *apply_at* *f* (*L* # *ds*) (*Node* *l* *a* *r*) = *Node* (*apply_at* *f* *ds* *l*) *a* *r*
| *apply_at* *f* (*R* # *ds*) (*Node* *l* *a* *r*) = *Node* *l* *a* (*apply_at* *f* *ds* *r*)

fun *apply_ats* :: ('*a* tree \Rightarrow '*a* tree) \Rightarrow *pos list* \Rightarrow '*a* tree \Rightarrow '*a* tree **where**

$apply_ats _ [] \ t = t \mid$
 $apply_ats \ f \ (p \# ps) \ t = apply_ats \ f \ ps \ (apply_at \ f \ p \ t)$

lemma *apply_ats_append*:
 $apply_ats \ f \ (ps_1 \ @ \ ps_2) \ t = apply_ats \ f \ ps_2 \ (apply_ats \ f \ ps_1 \ t)$
by (*induction ps₁ arbitrary: t*) *auto*

abbreviation *rotRs* $\equiv apply_ats \ rotR$

abbreviation *rotLs* $\equiv apply_ats \ rotL$

lemma *apply_ats_map_R*: $apply_ats \ f \ (map \ ((\#) \ R) \ ps) \ \langle l, a, r \rangle = Node$
 $l \ a \ (apply_ats \ f \ ps \ r)$
by(*induction ps arbitrary: r*) *auto*

lemma *inorder_rotRs_poss*: $inorder \ (rotRs \ (rotR_poss \ t) \ t) = inorder \ t$
apply(*induction t rule: rotR_poss.induct*)
apply(*auto simp: apply_ats_map_R*)
done

lemma *is_list_rotRs*: $is_list \ (rotRs \ (rotR_poss \ t) \ t)$
apply(*induction t rule: rotR_poss.induct*)
apply(*auto simp: apply_ats_map_R*)
done

lemma *is_list_rotRs_ps_t*: $is_list \ (rotRs \ ps \ t) \longrightarrow length \ ps \leq length \ (rotR_poss \ t)$
quickcheck[*expect=counterexample*]
oops

lemma *length_rotRs_poss*: $length \ (rotR_poss \ t) = size \ t - rlen \ t$
proof(*induction t rule: rotR_poss.induct*)
 case (*1 A a B b C*)
 then show *?case using rlen_le_size[of C] by simp*
qed auto

lemma *is_list_inorder_same*:
 $is_list \ t1 \implies is_list \ t2 \implies inorder \ t1 = inorder \ t2 \implies t1 = t2$
proof(*induction t1 arbitrary: t2*)
 case *Leaf*
 then show *?case by simp*
next
 case *Node*
 then show *?case by (cases t2) simp_all*
qed

```

lemma rot_id: rotLs (rev (rotR_poss t)) (rotRs (rotR_poss t) t) = t
apply(induction t rule: rotR_poss.induct)
apply(auto simp: apply_ats_map_R rev_map apply_ats_append)
done

corollary tree_to_tree_rotations: assumes inorder t1 = inorder t2
shows rotLs (rev (rotR_poss t2)) (rotRs (rotR_poss t1) t1) = t2
proof –
  have rotRs (rotR_poss t1) t1 = rotRs (rotR_poss t2) t2 (is ?L = ?R)
  by (simp add: assms inorder_rotRs_poss is_list_inorder_same is_list_rotRs)
  hence rotLs (rev (rotR_poss t2)) ?L = rotLs (rev (rotR_poss t2)) ?R
  by simp
  also have ... = t2 by(rule rot_id)
  finally show ?thesis .
qed

lemma size_rlen_better_ub: size t – rlen t ≤ size t – 1
by (cases t) auto

end

```

12 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  $\Rightarrow$   $'a$  list where
inorder Leaf = [] |
inorder (Node l (a,__) r) = inorder l @ a # inorder r

```

```

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

```

```

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

```

```

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

```

```

lemma eq_set_tree_empty[simp]: set_tree t = {}  $\longleftrightarrow$  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

13 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

14 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

```

14.1 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 $\text{fst } p \leq \text{snd } p$:

```

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

```

More precisely, $'a \text{ ivl}$ 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  $\Rightarrow$  'a where
  low p = fst (Rep_ivl p)

```

```

definition high :: 'a::linorder ivl  $\Rightarrow$  '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  $\implies$  high p = high q  $\implies$  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)

14.2 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,order_bot}) ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
⇒ 'a **where**
max3 a l r = max (high a) (max (max_hi l) (max_hi r))

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 l 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.prem1 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.prem1 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.prem1 Nl by auto
      obtain p2 where p2: p2 ∈ set_tree r high p2 = max_hi r
      using N.IH(2) N.prem1 Nr by auto
      then show ?thesis
      using p1 p2 N.prem1 by (auto simp add: max3_def max_def)
    qed
  qed
qed

```

14.3 Insertion and Deletion

definition node **where**

[simp]: node l a r = Node l (a, max3 a l 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) \implies 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 \implies 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* \Rightarrow '*a* *ivl* \times '*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* \Rightarrow '*a* *ivl_tree* \Rightarrow '*a* *ivl_tree* **where**

delete x Leaf = Leaf |

delete x (Node l (a, m) r) =

(*case cmp x a of*

LT \Rightarrow node (delete x l) a r |

GT \Rightarrow node l a (delete x r) |

EQ \Rightarrow 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') \implies t \neq Leaf \implies 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) \implies 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*:

$\llbracket t \neq \text{Leaf}; \text{inv_max_hi } t \rrbracket \implies \text{inv_max_hi (snd (split_min t))}$

by (*induct t*) (*auto split: prod.splits*)

lemma *inv_max_hi_delete*:

inv_max_hi t \implies 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*)

14.4 Search

Does interval x overlap with any interval in the tree?

```
fun search :: 'a::{linorder,order_bot} ivl_tree  $\Rightarrow$  'a ivl  $\Rightarrow$  bool where
  search Leaf  $x$  = False |
  search (Node l (a, m) r)  $x$  =
    (if overlap  $x$  a then True
     else if  $l \neq \text{Leaf} \wedge \text{max\_hi } l \geq \text{low } x$  then search l  $x$ 
     else search r  $x$ )
```

lemma search_correct:

```
inv_max_hi t  $\impl$  sorted (inorder t)  $\impl$  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.premis by (auto simp: sorted_wrt_append)

have search_r: search r x = has_overlap (set_tree r) x

using Node.IH(2) Node.premis 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]: $l = \text{Leaf}$

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 $l \neq \text{Leaf}$

then show ?thesis

proof (cases max_hi l \geq low x)

case max_hi_l_ge: True

have inv_max_hi l

using Node.premis(1) by auto

then obtain p where p : $p \in \text{set_tree } l$ high $p = \text{max_hi } l$

```

    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: overlap_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.prem(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
qed

```

definition empty :: 'a ivl_tree where
 empty = Leaf

14.5 Specification

```

locale Interval_Set = Set +
  fixes has_overlap :: 't  $\Rightarrow$  'a::linorder ivl  $\Rightarrow$  bool
  assumes set_overlap: invar s  $\implies$  has_overlap s x = Interval_Tree.has_overlap
    (set s) x

fun invar :: ('a::{linorder,order_bot}) ivl_tree  $\Rightarrow$  bool where
invar t = (inv_max_hi t  $\wedge$  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

```

15 AVL Tree Implementation of Sets

```

theory AVL_Set_Code
imports
  Cmp
  Isin2
begin

```

15.1 Code

```

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

```

```

definition empty :: 'a tree_ht where
  empty = Leaf

```

```

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

```

```

definition node :: 'a tree_ht  $\Rightarrow$  'a  $\Rightarrow$  'a tree_ht  $\Rightarrow$  'a tree_ht where
  node l a r = Node l (a, max (ht l) (ht r) + 1) r

```

```

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

```

```

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

```



```

fun insert :: 'a::linorder  $\Rightarrow$  'a tree_ht  $\Rightarrow$  '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  $\Rightarrow$  Node l (a, n) r |
  LT  $\Rightarrow$  balL (insert x l) a r |
  GT  $\Rightarrow$  balR l a (insert x r))

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

```

15.2 Functional Correctness Proofs

Very different from the AFP/AVL proofs

15.2.1 Proofs for insert

lemma inorder_balL:
 $\text{inorder } (\text{balL } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$
by (auto simp: node_def balL_def split:tree.splits)

lemma inorder_balR:
 $\text{inorder } (\text{balR } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$
by (auto simp: node_def balR_def split:tree.splits)

theorem inorder_insert:
 $\text{sorted}(\text{inorder } t) \Longrightarrow \text{inorder}(\text{insert } x \ t) = \text{ins_list } x \ (\text{inorder } t)$
by (induct t)
 (auto simp: ins_list_simps inorder_balL inorder_balR)

15.2.2 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

```

15.3 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)

```

15.3.1 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:*

$\llbracket \text{avl } l; \text{avl } r; \text{height } l \neq \text{height } r + 2 \rrbracket \implies$
 $\text{height } (\text{balL } l \ a \ r) = 1 + \max (\text{height } l) (\text{height } r)$

by (*simp_all add: balL_def*)

lemma *height_balR2:*

$\llbracket \text{avl } l; \text{avl } r; \text{height } r \neq \text{height } l + 2 \rrbracket \implies$
 $\text{height } (\text{balR } l \ a \ r) = 1 + \max (\text{height } l) (\text{height } r)$

by (*simp_all add: balR_def*)

lemma *avl_balL:*

$\llbracket \text{avl } l; \text{avl } r; \text{height } r - 1 \leq \text{height } l \wedge \text{height } l \leq \text{height } r + 2 \rrbracket \implies$
 $\text{avl}(\text{balL } l \ a \ r)$

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

lemma *avl_balR:*

$\llbracket \text{avl } l; \text{avl } r; \text{height } l - 1 \leq \text{height } r \wedge \text{height } r \leq \text{height } l + 2 \rrbracket \implies$
 $\text{avl}(\text{balR } l \ a \ r)$

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

Insertion maintains the AVL property. Requires simultaneous proof.

theorem *avl_insert:*

$\text{avl } t \implies \text{avl}(\text{insert } x \ t)$

$\text{avl } t \implies \text{height } (\text{insert } x \ t) \in \{\text{height } t, \text{height } t + 1\}$

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 in-*

tro!: 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* \implies
 $avl(insert\ x\ t) \wedge height\ (insert\ x\ t) \in \{height\ t, height\ t + 1\}$
apply (*induction t rule: tree2_induct*)

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

15.3.2 Deletion maintains AVL balance

lemma *avl_split_max*:
 $\llbracket avl\ t; t \neq Leaf \rrbracket \implies$
 $avl\ (fst\ (split_max\ t)) \wedge$
 $height\ t \in \{height(fst\ (split_max\ t)), height(fst\ (split_max\ t)) + 1\}$
by(*induct t rule: split_max_induct*)
 $(auto\ simp: balL_def node_def max_absorb2 split!: prod.split\ if_split\ tree.split)$

Deletion maintains the AVL property:

theorem *avl_delete*:
 $avl\ t \implies avl(delete\ x\ t)$
 $avl\ t \implies height\ t \in \{height\ (delete\ x\ t), height\ (delete\ x\ t) + 1\}$
proof (*induct t rule: tree2_induct*)
case (*Node l a n r*)
case 1
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  $\text{height } r = \text{height } (\text{delete } x \ l) + 2$ )
      case False
      thus ?thesis using 2 Node(1,2)  $\langle x < a \rangle$  by(auto simp: balR_def)
    next
      case True
      thus ?thesis using height_balR[OF __ True, of a] 2 Node(1,2)  $\langle x < a \rangle$  by simp linarith
    qed
  next
  case False
  show ?thesis
  proof(cases  $\text{height } l = \text{height } (\text{delete } x \ r) + 2$ )
    case False
    thus ?thesis using 2 Node(3,4)  $\langle \neg x < a \rangle \langle x \neq a \rangle$  by(auto simp:
balL_def)
  next
    case True
    thus ?thesis
      using height_balL[OF __ True, of a] 2 Node(3,4)  $\langle \neg x < a \rangle \langle x \neq a \rangle$ 
      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:
   $avl \ t \implies avl(\text{delete } x \ t)$ 
   $avl \ t \implies \text{height } t \in \{\text{height } (\text{delete } x \ t), \text{height } (\text{delete } x \ t) + 1\}$ 
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

```

15.4 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

```

15.5 Height-Size Relation

Any AVL tree of height n has at least $\text{fib}(n+2)$ leaves:

```

theorem avl_fib_bound:
  avl t  $\implies \text{fib}(\text{height } t + 2) \leq \text{size1 } t$ 
proof (induction rule: tree2_induct)
  case (Node l a h r)
  have 1:  $\text{height } l + 1 \leq \text{height } r + 2$  and 2:  $\text{height } r + 1 \leq \text{height } l + 2$ 
    using Node.premis by auto
  have fib (max (height l) (height r) + 3)  $\leq \text{size1 } l + \text{size1 } r$ 
  proof cases
    assume height l  $\geq$  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.prem1 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.prem2 by simp linarith
qed

```

An exponential lower bound for *fib*:

lemma *fib_lowerbound*:


```

defines  $\varphi \equiv (1 + \sqrt{5}) / 2$ 
shows  $\text{real } (\text{fib}(n+2)) \geq \varphi^n$ 
proof (induction n rule: fib.induct)
  case 1
  then show ?case by simp
next
  case 2
  then show ?case by (simp add:  $\varphi\_def$  real_le_sqrt)
next
  case ( $3\ n$ )
  have  $\varphi^{\text{Suc } (\text{Suc } n)} = \varphi^2 * \varphi^n$ 
    by (simp add: field_simps power2_eq_square)
  also have  $\dots = (\varphi + 1) * \varphi^n$ 
    by (simp_all add:  $\varphi\_def$  power2_eq_square field_simps)
  also have  $\dots = \varphi^{\text{Suc } n} + \varphi^n$ 
    by (simp add: field_simps)
  also have  $\dots \leq \text{real } (\text{fib } (\text{Suc } n + 2)) + \text{real } (\text{fib } (n + 2))$ 
    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  $\varphi \equiv (1 + \sqrt{5}) / 2$ 
  assumes avl t
  shows  $\varphi^{\text{height } t} \leq \text{size1 } t$ 
proof –
  have  $\varphi^{\text{height } t} \leq \text{fib } (\text{height } t + 2)$ 
    unfolding  $\varphi\_def$  by(rule fib_lowerbound)
  also have  $\dots \leq \text{size1 } t$ 
    using avl_fib_bound[of t] assms by simp
  finally show ?thesis .
qed

```

The height of an AVL tree is most $1 / \log 2 \varphi \approx 1.44$ times worse than $\log 2 (\text{real } (\text{size1 } t))$:

```

lemma avl_height_upperbound:
  defines  $\varphi \equiv (1 + \sqrt{5}) / 2$ 
  assumes avl t
  shows  $\text{height } t \leq (1 / \log 2 \varphi) * \log 2 (\text{size1 } t)$ 
proof –
  have  $\varphi > 0 \ \varphi > 1$  by(auto simp:  $\varphi\_def$  pos_add_strict)
  hence  $\text{height } t = \log \varphi (\varphi^{\text{height } t})$  by(simp add: log_nat_power)
  also have  $\dots \leq \log \varphi (\text{size1 } t)$ 

```

```

    using avl_size_lowerbound[OF assms(2), folded  $\varphi\_def$ ]  $\langle 1 < \varphi \rangle$ 
    by (simp add: le_log_of_power)
  also have ... =  $(1/\log 2 \ \varphi) * \log 2 \ (size1 \ t)$ 
    by (simp add: log_base_change[of 2  $\varphi$ ])
  finally show ?thesis .
qed

end

```

16 Function *lookup* for Tree2

```

theory Lookup2
imports
  Tree2
  Cmp
  Map_Specs
begin

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

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

```

17 AVL Tree Implementation of Maps

```

theory AVL_Map
imports
  AVL_Set
  Lookup2
begin

fun update ::  $'a::linorder \Rightarrow 'b \Rightarrow ('a * 'b) \text{ tree\_ht} \Rightarrow ('a * 'b) \text{ tree\_ht}$  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  $\Rightarrow Node \ l \ ((x,y), h) \ r$  |
  LT  $\Rightarrow balL \ (update \ x \ y \ l) \ (a,b) \ r$  |

```

$GT \Rightarrow \text{balR } l \ (a,b) \ (\text{update } x \ y \ r))$

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

17.1 Functional Correctness

theorem *inorder_update*:

$\text{sorted1}(\text{inorder } t) \Longrightarrow \text{inorder}(\text{update } x \ y \ t) = \text{upd_list } x \ y \ (\text{inorder } t)$
by (*induct* t) (*auto simp: upd_list_simps inorder_balL inorder_balR*)

theorem *inorder_delete*:

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

17.2 AVL invariants

17.2.1 Insertion maintains AVL balance

theorem *avl_update*:

assumes *avl* t
shows *avl*(*update* x y t)
 $(\text{height } (\text{update } x \ y \ t) = \text{height } t \vee \text{height } (\text{update } x \ y \ t) = \text{height } t + 1)$
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


```

      assume ?B with True 1 eq2(4) ⟨¬x < a⟩ show ?thesis by (simp)
arith
      qed
      qed
      qed
      qed
qed simp_all

```

17.2.2 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  $\langle x < a \rangle$  show ?thesis by(auto simp:
balR_def)
    next
      case True
      thus ?thesis using height_balR[OF __ True, of ab] 2 Node(1,2)  $\langle x$ 
 $< a \rangle$  by simp linarith
    qed
  next
    case False
    show ?thesis
    proof(cases height l = height (delete x r) + 2)
      case False with Node 1  $\langle \neg x < a \rangle$   $\langle x \neq a \rangle$  show ?thesis by(auto
simp: balL_def)
    next
      case True
      thus ?thesis
      using height_balL[OF __ True, of ab] 2 Node(3,4)  $\langle \neg x < a \rangle$   $\langle x$ 
 $\neq a \rangle$  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

18 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* \Rightarrow *bool* **where**

avl Leaf = *True* |

avl (Node l (a,b) r) =

((*case b of*

Bal \Rightarrow *height r* = *height l* |

Lh \Rightarrow *height l* = *height r* + 1 |

Rh \Rightarrow *height r* = *height l* + 1)

\wedge *avl l* \wedge *avl r*)

18.1 Code

fun *is_bal* **where**

is_bal (Node l (a,b) r) = (*b* = *Bal*)

fun *incr* **where**

incr t t' = (*t* = *Leaf* \vee *is_bal t* \wedge \neg *is_bal t'*)

fun *rot2* **where**

rot2 A a B c C = (*case B of*

(*Node B₁ (b, bb) B₂*) \Rightarrow

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* \Rightarrow *'a* \Rightarrow *bal* \Rightarrow *'a tree_bal* \Rightarrow *'a tree_bal* **where**

balL AB c bc C = (*case bc of*

Bal \Rightarrow *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 ∧ incr t' 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 incr 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 incr 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

```


$(a, ba) \ r^{\wedge}$

18.2 Proofs

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

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

declare *Let_def* [*simp*]

18.2.1 Proofs about insertion

lemma *avl_insert*: *avl t* \implies
 $avl(insert\ x\ t) \wedge$
 $height(insert\ x\ t) = height\ t + (if\ incr\ t\ (insert\ x\ t)\ then\ 1\ else\ 0)$
by (*induction x t rule: insert.induct*)(*auto split!: splits*)

The following two auxiliary lemma merely simplify the proof of *in-order_insert*.

lemma [*simp*]: $[] \neq ins_list\ x\ xs$
by(*cases xs*) *auto*

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

theorem *inorder_insert*:
 $\llbracket avl\ t; sorted(inorder\ t) \rrbracket \implies inorder(insert\ x\ t) = ins_list\ x\ (inorder\ t)$
by (*induction t*) (*auto simp: ins_list_simps split!: splits*)

18.2.2 Proofs about deletion

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

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

lemma *height_1_iff*: *avl t* $\implies height\ t = Suc\ 0 \longleftrightarrow (\exists x. t = Node\ Leaf\ (x, Bal)\ Leaf)$

by(cases t) (auto split: splits prod.splits)

lemma avl_split_max:

$\llbracket \text{split_max } t = (t', a); \text{avl } t; t \neq \text{Leaf} \rrbracket \implies$
 $\text{avl } t' \wedge \text{height } t = \text{height } t' + (\text{if incr } t' \text{ then } 1 \text{ else } 0)$

proof (induction t arbitrary: t' a rule: split_max_induct)

qed (auto simp: max_absorb1 max_absorb2 height_1_iff split!: splits prod.splits)

lemma avl_delete: avl t \implies

$\text{avl } (\text{delete } x \ t) \wedge$
 $\text{height } t = \text{height } (\text{delete } x \ t) + (\text{if decr } t \ (\text{delete } x \ t) \text{ then } 1 \text{ else } 0)$

proof (induction x t rule: delete_induct)

qed (auto simp: max_absorb1 max_absorb2 height_1_iff dest: avl_split_max split!: splits prod.splits)

lemma inorder_split_maxD:

$\llbracket \text{split_max } t = (t', a); t \neq \text{Leaf}; \text{avl } t \rrbracket \implies$
 $\text{inorder } t' @ [a] = \text{inorder } t$

proof (induction t arbitrary: t' rule: split_max_induct)

qed (auto split!: splits prod.splits)

lemma neq_Leaf_if_height_neq_0: height t $\neq 0 \implies t \neq \text{Leaf}$

by auto

lemma split_max_Leaf: $\llbracket t \neq \text{Leaf}; \text{avl } t \rrbracket \implies \text{split_max } t = (\langle \rangle, x) \longleftrightarrow$
 $t = \text{Node Leaf } (x, \text{Bal}) \text{ Leaf}$

by(cases t) (auto split: splits prod.splits)

theorem inorder_delete:

$\llbracket \text{avl } t; \text{sorted}(\text{inorder } t) \rrbracket \implies \text{inorder } (\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$

proof (induction t rule: tree2_induct)

case Leaf

then show ?case **by** auto

next

case (Node x1 a b x3)

then show ?case

by (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)

qed

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_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
```

19 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  $\Rightarrow$  bool where
avl Leaf = True |
avl (Node l (a,b) r) =
  ((case b of
```

$$\begin{aligned}
& Bal \Rightarrow height\ r = height\ l \mid \\
& Lh \Rightarrow height\ l = height\ r + 1 \mid \\
& Rh \Rightarrow height\ r = height\ l + 1) \\
& \wedge\ avl\ l \wedge\ avl\ r)
\end{aligned}$$

19.1 Code

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

type_synonym 'a tree_bal2 = 'a tree_bal alt

fun tree :: 'a alt \Rightarrow 'a **where**
 tree(Same t) = t |
 tree(Diff t) = t

fun rot2 **where**
 rot2 A a B c C = (case B of
 (Node B₁ (b, bb) B₂) \Rightarrow
 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_bal2 \Rightarrow 'a \Rightarrow bal \Rightarrow 'a tree_bal \Rightarrow 'a tree_bal2 **where**
 balL AB' c bc C = (case AB' of
 Same AB \Rightarrow Same (Node AB (c,bc) C) |
 Diff AB \Rightarrow (case bc of
 Bal \Rightarrow Diff (Node AB (c,Lh) C) |
 Rh \Rightarrow Same (Node AB (c,Bal) C) |
 Lh \Rightarrow (case AB of
 Node A (a,Lh) B \Rightarrow Same(Node A (a,Bal) (Node B (c,Bal) C)) |
 Node A (a,Rh) B \Rightarrow Same(rot2 A a B c C))))

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

fun ins :: 'a::linorder \Rightarrow 'a tree_bal \Rightarrow '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 balR :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal2 ⇒ 'a tree_bal2 where
balR 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 balL :: 'a tree_bal2 ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal2 where
balL 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 (balR 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 balL l' a' ba r |
    LT ⇒ balL (del x l) a ba r |
    GT ⇒ balR l a ba (del x r))

```

definition $delete :: 'a::linorder \Rightarrow 'a\ tree_bal \Rightarrow 'a\ tree_bal$ **where**
 $delete\ x\ t = tree(del\ x\ t)$

lemmas $split_max_induct = split_max.induct[case_names\ Node\ Leaf]$

lemmas $splits = if_splits\ tree.splits\ alt.splits\ bal.splits$

19.2 Proofs

19.2.1 Proofs about insertion

lemma $avl_ins_case: avl\ t \Longrightarrow case\ ins\ x\ t\ of$
 $Same\ t' \Rightarrow avl\ t' \wedge height\ t' = height\ t \mid$
 $Diff\ t' \Rightarrow avl\ t' \wedge height\ t' = height\ t + 1 \wedge$
 $(\forall l\ a\ r. t' = Node\ l\ (a, Bal)\ r \longrightarrow a = x \wedge l = Leaf \wedge r = Leaf)$
by $(induction\ x\ t\ rule: ins.induct)\ (auto\ simp: max_absorb1\ split!: splits)$

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

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 $inorder_ins:$
 $\llbracket avl\ t; sorted(inorder\ t) \rrbracket \Longrightarrow inorder(tree(ins\ x\ t)) = ins_list\ x\ (inorder\ t)$
by $(induction\ t)\ (auto\ simp: ins_list_simps\ split!: splits)$

19.2.2 Proofs about deletion

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

lemma $inorder_baldR:$
 $\llbracket ba = Lh \longrightarrow l \neq Leaf; avl\ l \rrbracket$
 $\Longrightarrow inorder\ (tree(baldR\ l\ a\ ba\ r)) = inorder\ l @ a \# inorder\ (tree\ r)$

by (*auto split: splits*)

lemma *avl_split_max*:

[[*split_max* $t = (t', a)$; *avl* t ; $t \neq \text{Leaf}$]] \implies *case* t' of
Same $t' \implies \text{avl } t' \wedge \text{height } t = \text{height } t' \mid$
Diff $t' \implies \text{avl } t' \wedge \text{height } t = \text{height } t' + 1$

proof (*induction* t *arbitrary*: t' *a rule*: *split_max_induct*)

qed (*auto simp: max_def split!: splits prod.splits*)

lemma *avl_del_case*: *avl* $t \implies$ *case* *del* x t of

Same $t' \implies \text{avl } t' \wedge \text{height } t = \text{height } t' \mid$
Diff $t' \implies \text{avl } t' \wedge \text{height } t = \text{height } t' + 1$

proof (*induction* x t *rule*: *del.induct*)

qed (*auto simp: max_absorb1 max_absorb2 dest: avl_split_max split!: splits prod.splits*)

corollary *avl_delete*: *avl* $t \implies \text{avl}(\text{delete } x \ t)$

using *avl_del_case*[of t x] **by**(*simp add: delete_def split: splits*)

lemma *inorder_split_maxD*:

[[*split_max* $t = (t', a)$; $t \neq \text{Leaf}$; *avl* t]] \implies
inorder (*tree* t') @ [a] = *inorder* t

proof (*induction* t *arbitrary*: t' *rule*: *split_max.induct*)

qed (*auto split!: splits prod.splits*)

lemma *neq_Leaf_if_height_neq_0*[*simp*]: *height* $t \neq 0 \implies t \neq \text{Leaf}$

by *auto*

theorem *inorder_del*:

[[*avl* t ; *sorted*(*inorder* t)]] \implies *inorder* (*tree*(*del* x t)) = *del_list* x (*inorder* t)

proof (*induction* t *rule*: *tree2_induct*)

case *Leaf*

then show ?*case* **by** *simp*

next

case (*Node* $x1$ a b $x3$)

then show ?*case*

by (*auto simp: del_list_simps inorder_balD inorder_balR avl_delete inorder_split_maxD*

simp del: balD.simps split!: splits prod.splits)

qed

19.2.3 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
```

20 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 $hbt :: 'a \text{ tree_ht} \Rightarrow \text{bool}$ **where**
 $hbt \text{ Leaf} = \text{True} \mid$
 $hbt (\text{Node } l (a, n) r) =$
 $(\text{abs}(\text{int}(\text{height } l) - \text{int}(\text{height } r)) \leq \text{int}(m) \wedge$
 $n = \text{max}(\text{height } l) (\text{height } r) + 1 \wedge hbt \ l \wedge hbt \ r)$

fun $ht :: 'a \text{ tree_ht} \Rightarrow \text{nat}$ **where**
 $ht \text{ Leaf} = 0 \mid$
 $ht (\text{Node } l (a, n) r) = n$

definition $node :: 'a \text{ tree_ht} \Rightarrow 'a \Rightarrow 'a \text{ tree_ht} \Rightarrow 'a \text{ tree_ht}$ **where**
 $node \ l \ a \ r = \text{Node } l (a, \text{max}(\text{ht } l) (\text{ht } r) + 1) \ r$

definition $balL :: 'a \text{ tree_ht} \Rightarrow 'a \Rightarrow 'a \text{ tree_ht} \Rightarrow 'a \text{ tree_ht}$ **where**
 $balL \ AB \ b \ C =$
 $(\text{if } ht \ AB = ht \ C + m + 1 \text{ then}$
 $\text{case } AB \text{ of}$
 $\text{Node } A (a, _) \ B \Rightarrow$
 $\text{if } ht \ A \geq ht \ B \text{ then } node \ A \ a (node \ B \ b \ C)$
 else
 $\text{case } B \text{ of}$
 $\text{Node } B_1 (ab, _) \ B_2 \Rightarrow node (node \ A \ a \ B_1) \ ab (node \ B_2 \ b \ C)$
 $\text{else } node \ AB \ b \ C)$

definition $balR :: 'a \text{ tree_ht} \Rightarrow 'a \Rightarrow 'a \text{ tree_ht} \Rightarrow 'a \text{ tree_ht}$ **where**
 $balR \ A \ a \ BC =$
 $(\text{if } ht \ BC = ht \ A + m + 1 \text{ then}$
 $\text{case } BC \text{ of}$
 $\text{Node } B (b, _) \ C \Rightarrow$
 $\text{if } ht \ B \leq ht \ C \text{ then } node (node \ A \ a \ B) \ b \ C$
 else
 $\text{case } B \text{ of}$
 $\text{Node } B_1 (ab, _) \ B_2 \Rightarrow node (node \ A \ a \ B_1) \ ab (node \ B_2 \ b \ C)$
 $\text{else } node \ A \ a \ BC)$

fun $insert :: 'a::\text{linorder} \Rightarrow 'a \text{ tree_ht} \Rightarrow 'a \text{ tree_ht}$ **where**
 $insert \ x \ \text{Leaf} = \text{Node } \text{Leaf} (x, 1) \ \text{Leaf} \mid$
 $insert \ x (\text{Node } l (a, n) r) = (\text{case } cmp \ x \ a \text{ of}$
 $EQ \Rightarrow \text{Node } l (a, n) \ r \mid$
 $LT \Rightarrow balL (insert \ x \ l) \ a \ r \mid$

$GT \Rightarrow \text{balR } l \ a \ (\text{insert } x \ r)$

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

20.1 Functional Correctness Proofs

20.1.1 Proofs for insert

lemma inorder_balL:
 $\text{inorder } (\text{balL } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$
by (auto simp: node_def balL_def split:tree.splits)

lemma inorder_balR:
 $\text{inorder } (\text{balR } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$
by (auto simp: node_def balR_def split:tree.splits)

theorem inorder_insert:
 $\text{sorted}(\text{inorder } t) \Longrightarrow \text{inorder}(\text{insert } x \ t) = \text{ins_list } x \ (\text{inorder } t)$
by (induct t)
 (auto simp: ins_list_simps inorder_balL inorder_balR)

20.1.2 Proofs for delete

lemma inorder_split_maxD:
 $\llbracket \text{split_max } t = (t', a); t \neq \text{Leaf} \rrbracket \Longrightarrow$
 $\text{inorder } t' \ @ \ [a] = \text{inorder } t$
by(induction t arbitrary: t' rule: split_max.induct)
 (auto simp: inorder_balL split: if_splits prod.splits tree.split)

theorem inorder_delete:
 $\text{sorted}(\text{inorder } t) \Longrightarrow \text{inorder } (\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by(induction t)

(*auto simp: del_list_simps inorder_balL inorder_balR inorder_split_maxD split: prod.splits*)

20.2 Invariant preservation

20.2.1 Insertion maintains balance

declare *Let_def* [*simp*]

lemma *ht_height*[*simp*]: $hbt\ t \implies ht\ t = height\ t$
by (*cases t rule: tree2_cases*) *simp_all*

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

lemma *height_balL*:
 $\llbracket hbt\ l; hbt\ r; height\ l = height\ r + m + 1 \rrbracket \implies$
 $height\ (balL\ l\ a\ r) \in \{height\ r + m + 1, height\ r + m + 2\}$
by (*auto simp: node_def balL_def split: tree.split*)

lemma *height_balR*:
 $\llbracket hbt\ l; hbt\ r; height\ r = height\ l + m + 1 \rrbracket \implies$
 $height\ (balR\ l\ a\ r) \in \{height\ l + m + 1, height\ l + m + 2\}$
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*:
 $\llbracket hbt\ l; hbt\ r; height\ l \neq height\ r + m + 1 \rrbracket \implies$
 $height\ (balL\ l\ a\ r) = 1 + \max\ (height\ l)\ (height\ r)$
by (*simp_all add: balL_def*)

lemma *height_balR2*:
 $\llbracket hbt\ l; hbt\ r; height\ r \neq height\ l + m + 1 \rrbracket \implies$
 $height\ (balR\ l\ a\ r) = 1 + \max\ (height\ l)\ (height\ r)$
by (*simp_all add: balR_def*)

lemma *hbt_balL*:
 $\llbracket hbt\ l; hbt\ r; height\ r - m \leq height\ l \wedge height\ l \leq height\ r + m + 1 \rrbracket$
 $\implies hbt(balL\ l\ a\ r)$
by(*auto simp: balL_def node_def max_def split!: if_splits tree.split*)

lemma *hbt_balR*:
 $\llbracket hbt\ l; hbt\ r; height\ l - m \leq height\ r \wedge height\ r \leq height\ l + m + 1 \rrbracket$
 $\implies hbt(balR\ l\ a\ r)$

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 \implies *hbt*(*insert x t*)

hbt t \implies *height* (*insert x t*) \in {*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)* $\langle x \neq a \rangle$ **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)* $\langle x < a \rangle$ **show** ?*thesis* **by** (*auto simp:*

height_balL2)

next

case *True*

hence (*height (balL (insert x l) a r) = height r + m + 1*) \vee

(*height (balL (insert x l) a r) = height r + m + 2*) (**is** ?*A* \vee ?*B*)

using 2 *Node(1,2)* *height_balL[OF _ _ True]* **by** *simp*

thus ?*thesis*

proof

assume ?*A* **with** 2 *Node(2)* *True* $\langle x < a \rangle$ **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  $\implies$ 
  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

```

20.2.2 Deletion maintains balance

```

lemma hbt_split_max:
  [ hbt t; t  $\neq$  Leaf ]  $\implies$ 
  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 \implies *hbt*(*delete x t*)

hbt t \implies *height t* $\in \{\text{height } (\text{delete } x \ t), \text{height } (\text{delete } x \ t) + 1\}$

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* $\langle x < a \rangle$ **show** ?*thesis* **by**(*auto simp: balR_def*)

next

case *True*

hence (*height (balR (delete x l) a r) = height (delete x l) + m + 1*)

\vee

height (balR (delete x l) a r) = height (delete x l) + m + 2 (**is** ?*A*

\vee ?*B*)

using *Node 2 height_balR*[*OF* __ *True*] **by** *simp*

thus ?*thesis*

proof

assume ?*A* **with** $\langle x < a \rangle$ *Node 2* **show** ?*thesis* **by**(*auto simp: balR_def split!: if_splits*)

next

assume ?*B* **with** $\langle x < a \rangle$ *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  $\langle \neg x < a \rangle \langle x \neq a \rangle$  show ?thesis by(auto
simp: balL_def)
  next
    case True
    hence (height (balL l a (delete x r)) = height (delete x r) + m + 1)
 $\vee$ 
    height (balL l a (delete x r)) = height (delete x r) + m + 2 (is ?A
 $\vee$  ?B)
    using Node 2 height_balL[OF _ _ True] by simp
    thus ?thesis
    proof
      assume ?A with  $\langle \neg x < a \rangle \langle x \neq a \rangle$  Node 2 show ?thesis by(auto
simp: balL_def split: if_splits)
      next
        assume ?B with  $\langle \neg x < a \rangle \langle x \neq a \rangle$  Node 2 show ?thesis by(auto
simp: balL_def split: if_splits)
      qed
    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  $\implies$  hbt(delete x t)
  hbt t  $\implies$  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

```

20.3 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

```

21 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  $\equiv$  Node l (a, Red) r
abbreviation B where B l a r  $\equiv$  Node l (a, Black) r

```



```

fun baliL :: 'a rbt  $\Rightarrow$  'a  $\Rightarrow$  'a rbt  $\Rightarrow$  '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  $\Rightarrow$  'a  $\Rightarrow$  'a rbt  $\Rightarrow$  '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  $\Rightarrow$  'a rbt  $\Rightarrow$  'a rbt where
  paint c Leaf = Leaf |
  paint c (Node l (a,_) r) = Node l (a,c) r

```

```

fun baldL :: 'a rbt  $\Rightarrow$  'a  $\Rightarrow$  'a rbt  $\Rightarrow$  '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  $\Rightarrow$  'a  $\Rightarrow$  'a rbt  $\Rightarrow$  '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  $\Rightarrow$  'a rbt  $\Rightarrow$  '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  $\Rightarrow$  (R (R t1 a u2) b (R u3 c t4)) |
      t23  $\Rightarrow$  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  $\Rightarrow$  R (B t1 a u2) b (B u3 c t4) |
      t23  $\Rightarrow$  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

```

22 Red-Black Tree Implementation of Sets

theory *RBTree_Set*

imports

Complex_Main

RBTree

Cmp

Isin2

begin

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

empty = *Leaf*

fun *ins* :: 'a::linorder \Rightarrow 'a rbt \Rightarrow 'a rbt **where**

ins *x* *Leaf* = *R Leaf x Leaf* |

ins *x* (*B l a r*) =

(*case cmp x a of*

LT \Rightarrow *baliL (ins x l) a r* |

GT \Rightarrow *baliR l a (ins x r)* |

EQ \Rightarrow *B l a r*) |

ins *x* (*R l a r*) =

(*case cmp x a of*

LT \Rightarrow *R (ins x l) a r* |

GT \Rightarrow *R l a (ins x r)* |

EQ \Rightarrow *R l a r*)

definition *insert* :: 'a::linorder \Rightarrow 'a rbt \Rightarrow 'a rbt **where**

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

fun *color* :: 'a rbt \Rightarrow *color* **where**

color *Leaf* = *Black* |

color (*Node* _ (_, *c*) _) = *c*

fun *del* :: 'a::linorder \Rightarrow 'a rbt \Rightarrow 'a rbt **where**

del *x* *Leaf* = *Leaf* |

del *x* (*Node l (a, _) r*) =

(*case cmp x a of*

LT \Rightarrow *if l \neq Leaf \wedge color l = Black*

then baldL (del x l) a r else R (del x l) a r |

GT \Rightarrow *if r \neq Leaf \wedge color r = Black*

then baldR l a (del x r) else R l a (del x r) |

EQ \Rightarrow *join l r*)

definition *delete* :: 'a::linorder \Rightarrow 'a rbt \Rightarrow 'a rbt **where**

delete x t = paint Black (del x t)

22.1 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) \implies 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) \implies 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) \implies 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*:
 $\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by (*auto simp: delete_def inorder_del inorder_paint*)

22.2 Structural invariants

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

The proofs are due to Markus Reiter and Alexander Krauss.

fun *bheight* :: 'a rbt \Rightarrow 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 \Rightarrow bool **where**
invc Leaf = True |
invc (Node *l* (*a*,*c*) *r*) =
 $((c = \text{Red} \longrightarrow \text{color } l = \text{Black} \wedge \text{color } r = \text{Black}) \wedge \text{invc } l \wedge \text{invc } r)$

Weaker version:

abbreviation *invc2* :: 'a rbt \Rightarrow bool **where**
invc2 *t* \equiv *invc*(*paint* Black *t*)

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

lemma *invc2I*: *invc t* \implies *invc2 t*
by (*cases t rule: tree2_cases*) *simp+*

definition *rbt* :: 'a rbt \Rightarrow bool **where**
rbt t = (*invc t* \wedge *invh t* \wedge *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* \implies *invh* (*paint c t*)
by (*cases t*) *auto*

lemma *invc_baliL*:
 $\llbracket \text{invc2 } l; \text{ invc } r \rrbracket \implies \text{invc } (\text{baliL } l \ a \ r)$

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

lemma *invc_baliR:*

$\llbracket \text{invc } l; \text{ invc2 } r \rrbracket \implies \text{invc } (\text{baliR } l \text{ a } r)$

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

lemma *bheight_baliL:*

$\text{bheight } l = \text{bheight } r \implies \text{bheight } (\text{baliL } l \text{ a } r) = \text{Suc } (\text{bheight } l)$

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

lemma *bheight_baliR:*

$\text{bheight } l = \text{bheight } r \implies \text{bheight } (\text{baliR } l \text{ a } r) = \text{Suc } (\text{bheight } l)$

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

lemma *invh_baliL:*

$\llbracket \text{invh } l; \text{ invh } r; \text{ bheight } l = \text{bheight } r \rrbracket \implies \text{invh } (\text{baliL } l \text{ a } r)$

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

lemma *invh_baliR:*

$\llbracket \text{invh } l; \text{ invh } r; \text{ bheight } l = \text{bheight } r \rrbracket \implies \text{invh } (\text{baliR } l \text{ a } r)$

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

All in one:

lemma *inv_baliR:* $\llbracket \text{invh } l; \text{ invh } r; \text{ invc } l; \text{ invc2 } r; \text{ bheight } l = \text{bheight } r \rrbracket \implies \text{invc } (\text{baliR } l \text{ a } r) \wedge \text{invh } (\text{baliR } l \text{ a } r) \wedge \text{bheight } (\text{baliR } l \text{ a } r) = \text{Suc } (\text{bheight } l)$

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

lemma *inv_baliL:* $\llbracket \text{invh } l; \text{ invh } r; \text{ invc2 } l; \text{ invc } r; \text{ bheight } l = \text{bheight } r \rrbracket \implies \text{invc } (\text{baliL } l \text{ a } r) \wedge \text{invh } (\text{baliL } l \text{ a } r) \wedge \text{bheight } (\text{baliL } l \text{ a } r) = \text{Suc } (\text{bheight } l)$

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

22.2.1 Insertion

lemma *invc_ins:* $\text{invc } t \longrightarrow \text{invc2 } (\text{ins } x \text{ } t) \wedge (\text{color } t = \text{Black} \longrightarrow \text{invc } (\text{ins } x \text{ } t))$

by (*induct x t rule: ins.induct*) (*auto simp: invc_baliL invc_baliR invc2I*)

lemma *invh_ins:* $\text{invh } t \implies \text{invh } (\text{ins } x \text{ } t) \wedge \text{bheight } (\text{ins } x \text{ } t) = \text{bheight } t$

by (*induct x t rule: ins.induct*)

(*auto simp: invh_baliL invh_baliR bheight_baliL bheight_baliR*)

theorem *rbt_insert:* $\text{rbt } t \implies \text{rbt } (\text{insert } x \text{ } t)$

by (*simp add: invc_ins invh_ins color_paint_Black invh_paint rbt_def insert_def*)

All in one:

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

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

22.2.2 Deletion

lemma *bheight_paint_Red*:
 $\text{color } t = \text{Black} \implies \text{bheight } (\text{paint Red } t) = \text{bheight } t - 1$
by (*cases t*) *auto*

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

lemma *invh_baldL_Black*:
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l + 1 = \text{bheight } r; \text{color } r = \text{Black} \rrbracket$
 $\implies \text{invh } (\text{baldL } l \ a \ r) \wedge \text{bheight } (\text{baldL } l \ a \ r) = \text{bheight } r$
by (*induct l a r rule: baldL.induct*) (*auto simp add: invh_baliR bheight_baliR*)

lemma *invc_baldL*: $\llbracket \text{invc2 } l; \text{invc } r; \text{color } r = \text{Black} \rrbracket \implies \text{invc } (\text{baldL } l \ a \ r)$
by (*induct l a r rule: baldL.induct*) (*simp_all add: invc_baliR*)

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

lemma *invh_baldR_invc*:
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l = \text{bheight } r + 1; \text{invc } l \rrbracket$
 $\implies \text{invh } (\text{baldR } l \ a \ r) \wedge \text{bheight } (\text{baldR } l \ a \ r) = \text{bheight } l$
by(*induct l a r rule: baldR.induct*)
(auto simp: invh_baliL bheight_baliL invh_paint bheight_paint_Red)

lemma *invc_baldR*: $\llbracket \text{invc } l; \text{invc2 } r; \text{color } l = \text{Black} \rrbracket \implies \text{invc } (\text{baldR } l \ a \ r)$

r)
by (*induct* l a r rule: *baldR.induct*) (*simp_all* add: *invc_baliL*)

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

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

lemma *invc_join*:
 $\llbracket \text{invc } l; \text{invc } r \rrbracket \implies$
 $(\text{color } l = \text{Black} \wedge \text{color } r = \text{Black} \longrightarrow \text{invc } (\text{join } l \ r)) \wedge \text{invc2 } (\text{join } l \ r)$
by (*induct* l r rule: *join.induct*)
(*auto simp*: *invc_baldL invc2I split: tree.splits color.splits*)

All in one:

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

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

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

lemma *neq_LeafD*: $t \neq \text{Leaf} \implies \exists l \ x \ c \ r. t = \text{Node } l \ (x, c) \ r$

by(*cases t rule: tree2_cases*) *auto*

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

theorem *rbt_delete*: $\text{rbt } t \implies \text{rbt } (\text{delete } x \ t)$
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

22.3 Height-Size Relation

lemma *rbt_height_bheight_if*: $\text{invc } t \implies \text{invh } t \implies$
 $\text{height } t \leq 2 * \text{bheight } t + (\text{if } \text{color } t = \text{Black} \text{ then } 0 \text{ else } 1)$
by(*induction t*) (*auto split: if_split_asm*)

lemma *rbt_height_bheight*: $\text{rbt } t \implies \text{height } t / 2 \leq \text{bheight } t$
by(*auto simp: rbt_def dest: rbt_height_bheight_if*)

lemma *bheight_size_bound*: $\text{invc } t \implies \text{invh } t \implies 2^{\text{bheight } t} \leq \text{size1 } t$

by (*induction t*) *auto*

lemma *bheight_le_min_height*: *invh t \implies bheight t \leq min_height t*
by (*induction t*) *auto*

lemma *rbt_height_le*: **assumes** *rbt t* **shows** *height t \leq 2 * log 2 (size1 t)*
proof –

have *2 powr (height t / 2) \leq 2 powr bheight t*
 using *rbt_height_bheight[OF assms]* **by** *simp*
 also have *... \leq size1 t* **using** *assms*
 by (*simp add: powr_realpow bheight_size_bound rbt_def*)
 finally have *2 powr (height t / 2) \leq size1 t .*
 hence *height t / 2 \leq log 2 (size1 t)*
 by (*simp add: le_log_iff size1_size del: divide_le_eq_numeral1(1)*)
 thus ?thesis **by** *simp*

qed

lemma *rbt_height_le2*: **assumes** *rbt t* **shows** *height t \leq 2 * log 2 (size1 t)*

proof –

have *height t \leq 2 * bheight t*
 using *rbt_height_bheight_if assms[simplified rbt_def]* **by** *fastforce*
 also have *... \leq 2 * min_height t*
 using *bheight_le_min_height assms[simplified rbt_def]* **by** *auto*
 also have *... \leq 2 * log 2 (size1 t)*
 using *le_log2_of_power min_height_size1* **by** *auto*
 finally show *?thesis* **by** *simp*

qed

end

23 Alternative Deletion in Red-Black Trees

theory *RBTree_Set2*
imports *RBTree_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 \Rightarrow 'a \times 'a rbt* **where**
split_min (Node l (a, _) r) =
 (*if l = Leaf then (a,r)*
 else let (x,l') = split_min l

$in\ (x, \text{if } color\ l = Black \text{ then } baldL\ l'\ a\ r \text{ else } R\ l'\ a\ r))$

```
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 *lets* 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)
```

23.1 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(delete 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) ⇒ inorder(delete x t) = del_list x (inorder t)
by (auto simp: delete_def inorder_del inorder_paint)
```

23.2 Structural invariants

```
lemma neq_Red[simp]: (c ≠ Red) = (c = Black)
by (cases c) auto
```

23.2.1 Deletion

```
lemma inv_split_min: [| split_min t = (x,t'); t ≠ Leaf; invh t; invc t |]
⇒
```

```

    invh t' ∧
    (color t = Red → bheight t' = bheight t ∧ invc t') ∧
    (color t = Black → bheight t' = bheight t - 1 ∧ 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 *lets* in *RBT_Set2.del* breaks it.

```

lemma inv_del: [ 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))
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: [ 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(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.prem 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.prem inv_baldL[of del x l] *

```

```

l by(auto)
  next
    assume color l = Red
    thus ?thesis using ⟨x < a⟩ 2.prem1 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.prem1 neq_LeafD[of r] inv_balDR[of
    _ del x r]
      by(auto split: if_split)

  next
    assume ¬ x > a
    show ?thesis using 2.prem1 ⟨¬ 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  $\gamma$  thus ?case by (simp add: rbt_delete)
qed

```

```

end

```

24 Red-Black Tree Implementation of Maps

```

theory RBT_Map

```

```

imports

```

```

  RBT_Set

```

```

  Lookup2

```

```

begin

```

```

fun upd :: 'a::linorder  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('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  $\Rightarrow$  balL (upd x y l) (a,b) r |

```

```

    GT  $\Rightarrow$  balR l (a,b) (upd x y r) |

```

```

    EQ  $\Rightarrow$  B l (x,y) r) |

```

```

  upd x y (R l (a,b) r) = (case cmp x a of

```

```

    LT  $\Rightarrow$  R (upd x y l) (a,b) r |

```

```

    GT  $\Rightarrow$  R l (a,b) (upd x y r) |

```

```

    EQ  $\Rightarrow$  R l (x,y) r)

```

```

definition update :: 'a::linorder  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt where

```

```

  update x y t = paint Black (upd x y t)

```

```

fun del :: 'a::linorder  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt where

```

```

  del x Leaf = Leaf |

```

```

  del x (Node l (ab, _) r) = (case cmp x (fst ab) of

```

```

    LT  $\Rightarrow$  if l  $\neq$  Leaf  $\wedge$  color l = Black

```

```

      then balL (del x l) ab r else R (del x l) ab r |

```

```

    GT  $\Rightarrow$  if r  $\neq$  Leaf  $\wedge$  color r = Black

```

```

      then balR l ab (del x r) else R l ab (del x r) |

```

```

    EQ  $\Rightarrow$  join l r)

```

```

definition delete :: 'a::linorder  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt where

```

```

  delete x t = paint Black (del x t)

```

24.1 Functional Correctness Proofs

```

lemma inorder_upd:

```

$sorted1(inorder\ t) \implies inorder(upd\ x\ y\ t) = upd_list\ x\ y\ (inorder\ t)$
by(*induction* $x\ y\ t$ *rule*: *upd.induct*)
 (auto simp: *upd_list_simps* *inorder_baliL* *inorder_baliR*)

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

lemma *del_list_id*: $\forall ab \in set\ ps.\ y < fst\ ab \implies x \leq y \implies del_list\ x\ ps = ps$
by(rule *del_list_idem*) auto

lemma *inorder_del*:
 $sorted1(inorder\ t) \implies inorder(del\ x\ t) = del_list\ x\ (inorder\ t)$
by(*induction* $x\ t$ *rule*: *del.induct*)
 (auto simp: *del_list_simps* *del_list_id* *inorder_join* *inorder_baldL* *inorder_baldR*)

lemma *inorder_delete*:
 $sorted1(inorder\ t) \implies inorder(delete\ x\ t) = del_list\ x\ (inorder\ t)$
by(simp add: *delete_def* *inorder_del* *inorder_paint*)

24.2 Structural invariants

24.2.1 Update

lemma *invc_upd*: **assumes** *invc* t
shows $color\ t = Black \implies invc\ (upd\ x\ y\ t)\ invc2\ (upd\ x\ y\ t)$
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*: $rbt\ t \implies rbt\ (update\ x\ y\ t)$
by (simp add: *invc_upd*(2) *invh_upd*(1) *color_paint_Black* *invh_paint* *rbt_def* *update_def*)

24.2.2 Deletion

lemma *del_invc_invh*: $invh\ t \implies invc\ t \implies invh\ (del\ x\ t) \wedge$

```

    (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 _ ab c)
  have x = fst ab ∨ x < fst ab ∨ x > fst ab by auto
  thus ?case proof (elim disjE)
    assume x = fst ab
    with 2 show ?thesis
    by (cases c) (simp_all add: invh_join invc_join)
  next
    assume x < fst ab
    with 2 show ?thesis
    by (cases c)
      (auto simp: invh_baldL_invc invc_baldL invc2_baldL dest: neq_LeafD)
  next
    assume fst ab < x
    with 2 show ?thesis
    by (cases c)
      (auto simp: invh_baldR_invc invc_baldR invc2_baldR dest: neq_LeafD)
  qed
qed auto

theorem rbt_delete: rbt t ⇒ rbt (delete k t)
by (metis delete_def rbt_def color_paint_Black del_invc_invh 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

25 2-3 Trees

theory *Tree23*
imports *Main*
begin

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

datatype 'a *tree23* =
Leaf ($\langle \rangle$) |
Node2 'a *tree23* 'a 'a *tree23* ($\langle _, _, _ \rangle$) |
Node3 'a *tree23* 'a 'a *tree23* 'a 'a *tree23* ($\langle _, _, _, _, _ \rangle$)

fun *inorder* :: 'a *tree23* \Rightarrow 'a list **where**
inorder *Leaf* = [] |
inorder (*Node2* l a r) = *inorder* l @ a # *inorder* r |
inorder (*Node3* l a m b r) = *inorder* l @ a # *inorder* m @ b # *inorder* r

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

fun *height_tree23* :: 'a *tree23* \Rightarrow 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* \Rightarrow bool **where**
complete *Leaf* = *True* |
complete (*Node2* l _ r) = (*height* l = *height* r \wedge *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*: $complete\ t \implies 2^{\text{height } t} \leq size\ t + 1$
by (*induction t*) *auto*

end

26 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* \Rightarrow 'a \Rightarrow *bool* **where**

isin Leaf *x* = *False* |

isin (Node2 l a r) *x* =

(*case cmp x a of*

LT \Rightarrow *isin l x* |

EQ \Rightarrow *True* |

GT \Rightarrow *isin r x*) |

isin (Node3 l a m b r) *x* =

(*case cmp x a of*

LT \Rightarrow *isin l x* |

EQ \Rightarrow *True* |

GT \Rightarrow

(*case cmp x b of*

LT \Rightarrow *isin m x* |

EQ \Rightarrow *True* |

GT \Rightarrow *isin r x*))

datatype 'a *up_i* = *Eq_i* 'a *tree23* | *Of* 'a *tree23* 'a 'a *tree23*

fun *tree_i* :: 'a *up_i* \Rightarrow 'a *tree23* **where**

tree_i (Eq_i t) = *t* |

tree_i (Of l a r) = *Node2 l a r*

fun *ins* :: 'a::*linorder* \Rightarrow 'a *tree23* \Rightarrow 'a *up_i* **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
        Eqi l' => Eqi (Node2 l' a r) |
        Of l1 b l2 => Eqi (Node3 l1 b l2 a r)) |
    EQ => Eqi (Node2 l a r) |
    GT =>
      (case ins x r of
        Eqi r' => Eqi (Node2 l a r') |
        Of r1 b r2 => Eqi (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
        Eqi l' => Eqi (Node3 l' a m b r) |
        Of l1 c l2 => Of (Node2 l1 c l2) a (Node2 m b r)) |
    EQ => Eqi (Node3 l a m b r) |
    GT =>
      (case cmp x b of
        GT =>
          (case ins x r of
            Eqi r' => Eqi (Node3 l a m b r') |
            Of r1 c r2 => Of (Node2 l a m) b (Node2 r1 c r2)) |
        EQ => Eqi (Node3 l a m b r) |
        LT =>
          (case ins x m of
            Eqi m' => Eqi (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 = tree_i(ins x t)

datatype 'a up_d = Eq_d 'a tree23 | Uf 'a tree23

fun tree_d :: 'a up_d => 'a tree23 **where**
 tree_d (Eq_d t) = t |
 tree_d (Uf t) = t

```

fun node21 :: 'a up_d  $\Rightarrow$  'a  $\Rightarrow$  'a tree23  $\Rightarrow$  'a up_d where
node21 (Eq_d t1) a t2 = Eq_d(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) = Eq_d(Node2 (Node2 t1 a t2) b
(Node2 t3 c t4))

```

```

fun node22 :: 'a tree23  $\Rightarrow$  'a  $\Rightarrow$  'a up_d  $\Rightarrow$  'a up_d where
node22 t1 a (Eq_d t2) = Eq_d(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) = Eq_d(Node2 (Node2 t1 b t2) c
(Node2 t3 a t4))

```

```

fun node31 :: 'a up_d  $\Rightarrow$  'a  $\Rightarrow$  'a tree23  $\Rightarrow$  'a  $\Rightarrow$  'a tree23  $\Rightarrow$  'a up_d where
node31 (Eq_d t1) a t2 b t3 = Eq_d(Node3 t1 a t2 b t3) |
node31 (Uf t1) a (Node2 t2 b t3) c t4 = Eq_d(Node2 (Node3 t1 a t2 b t3)
c t4) |
node31 (Uf t1) a (Node3 t2 b t3 c t4) d t5 = Eq_d(Node3 (Node2 t1 a t2)
b (Node2 t3 c t4) d t5)

```

```

fun node32 :: 'a tree23  $\Rightarrow$  'a  $\Rightarrow$  'a up_d  $\Rightarrow$  'a  $\Rightarrow$  'a tree23  $\Rightarrow$  'a up_d where
node32 t1 a (Eq_d t2) b t3 = Eq_d(Node3 t1 a t2 b t3) |
node32 t1 a (Uf t2) b (Node2 t3 c t4) = Eq_d(Node2 t1 a (Node3 t2 b t3 c
t4)) |
node32 t1 a (Uf t2) b (Node3 t3 c t4 d t5) = Eq_d(Node3 t1 a (Node2 t2 b
t3) c (Node2 t4 d t5))

```

```

fun node33 :: 'a tree23  $\Rightarrow$  'a  $\Rightarrow$  'a tree23  $\Rightarrow$  'a  $\Rightarrow$  'a up_d  $\Rightarrow$  'a up_d where
node33 t1 a t2 b (Eq_d t3) = Eq_d(Node3 t1 a t2 b t3) |
node33 t1 a (Node2 t2 b t3) c (Uf t4) = Eq_d(Node2 t1 a (Node3 t2 b t3 c
t4)) |
node33 t1 a (Node3 t2 b t3 c t4) d (Uf t5) = Eq_d(Node3 t1 a (Node2 t2 b
t3) c (Node2 t4 d t5))

```

```

fun split_min :: 'a tree23  $\Rightarrow$  'a * 'a up_d where
split_min (Node2 Leaf a Leaf) = (a, Uf Leaf) |
split_min (Node3 Leaf a Leaf b Leaf) = (a, Eq_d(Node2 Leaf b Leaf)) |
split_min (Node2 l a r) = (let (x,l') = split_min l in (x, node21 l' a r)) |
split_min (Node3 l a m b r) = (let (x,l') = split_min l in (x, node31 l' a
m b r))

```

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

```

fun del :: 'a::linorder  $\Rightarrow$  'a tree23  $\Rightarrow$  'a up_d where
del x Leaf = Eq_d Leaf |

```

```

del x (Node2 Leaf a Leaf) =
  (if x = a then Uf Leaf else Eqd(Node2 Leaf a Leaf)) |
del x (Node3 Leaf a Leaf b Leaf) =
  Eqd(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 = tree_d(del x t)

26.1 Functional Correctness

26.1.1 Proofs for isin

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

26.1.2 Proofs for insert

lemma *inorder_ins*:
 sorted(*inorder* t) ⇒ *inorder*(tree_i(ins x t)) = ins_list x (*inorder* t)
by(*induction* t) (*auto simp: ins_list_simps split: up_i.splits*)

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

26.1.3 Proofs for delete

lemma *inorder_node21*: height r > 0 ⇒
inorder (tree_d (node21 l' a r)) = *inorder* (tree_d l') @ a # *inorder* r

by(*induct l' a r rule: node21.induct*) *auto*

lemma *inorder_node22: height l > 0 \implies*

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 \implies*

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 \implies*

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 \implies*

inorder (tree_d (node33 l a m b r')) = inorder l @ a # inorder m @ b #

inorder (tree_d r')

by(*induct l a m b r' rule: node33.induct*) *auto*

lemmas *inorder_nodes = inorder_node21 inorder_node22*

inorder_node31 inorder_node32 inorder_node33

lemma *split_minD:*

split_min t = (x, t') \implies complete t \implies height t > 0 \implies

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: \llbracket complete t ; sorted(inorder t) $\rrbracket \implies$*

inorder(tree_d (del x t)) = del_list x (inorder t)

by(*induction t rule: del.induct*)

(*auto simp: del_list_simps inorder_nodes split_minD split!: if_split prod.splits*)

lemma *inorder_delete: \llbracket complete t ; sorted(inorder t) $\rrbracket \implies$*

inorder(delete x t) = del_list x (inorder t)

by(*simp add: delete_def inorder_del*)

26.2 Completeness

26.2.1 Proofs for insert

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

```

fun  $h_i :: 'a \text{ up}_i \Rightarrow \text{nat}$  where
 $h_i (Eq_i t) = \text{height } t \mid$ 
 $h_i (Of l a r) = \text{height } l$ 

```

```

lemma complete_ins:  $\text{complete } t \Longrightarrow \text{complete } (\text{tree}_i(\text{ins } a \ t)) \wedge h_i(\text{ins } a \ t) = \text{height } t$ 
by (induct  $t$ ) (auto split!: if_split up_i.split)

```

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

```

inductive full ::  $\text{nat} \Rightarrow 'a \text{ tree23} \Rightarrow \text{bool}$  where
 $\text{full } 0 \text{ Leaf} \mid$ 
 $\llbracket \text{full } n \ l; \text{full } n \ r \rrbracket \Longrightarrow \text{full } (\text{Suc } n) (\text{Node2 } l \ p \ r) \mid$ 
 $\llbracket \text{full } n \ l; \text{full } n \ m; \text{full } n \ r \rrbracket \Longrightarrow \text{full } (\text{Suc } n) (\text{Node3 } l \ p \ m \ q \ r)$ 

```

```

inductive_cases full_elims:
 $\text{full } n \text{ Leaf}$ 
 $\text{full } n (\text{Node2 } l \ p \ r)$ 
 $\text{full } n (\text{Node3 } l \ p \ m \ q \ r)$ 

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma full_imp_complete:  $\text{full } n \ t \Longrightarrow \text{complete } t$ 
by (induct set: full, auto dest: full_imp_height)

```

```

lemma complete_imp_full:  $\text{complete } t \Longrightarrow \text{full } (\text{height } t) \ t$ 

```

by (*induct t, simp_all*)

lemma *complete_iff_full*: *complete t* \longleftrightarrow ($\exists n. \text{full } n \ t$)
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 *Eq_i 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 up_i* \Rightarrow *bool* **where**
full_i n (Eq_i 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* \implies *full_i n (ins a t)*
by (*induct rule: full.induct*) (*auto split: up_i.split*)

The *insert* operation preserves completeance.

lemma *complete_insert*: *complete t* \implies *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

26.3 Proofs for delete

fun *h_d* :: '*a up_d* \Rightarrow *nat* **where**
h_d (Eq_d t) = *height t* |
h_d (Uf t) = *height t* + 1

lemma *complete_tree_d_node21*:
 $\llbracket \text{complete } r; \text{ complete } (\text{tree}_d \ l'); \text{ height } r = h_d \ l' \rrbracket \implies \text{complete } (\text{tree}_d \ (\text{node21 } l' \ a \ r))$
by(*induct l' a r rule: node21.induct*) *auto*

lemma *complete_tree_d_node22*:
 $\llbracket \text{complete}(\text{tree}_d \ r'); \text{ complete } l; h_d \ r' = \text{height } l \rrbracket \implies \text{complete } (\text{tree}_d \ (\text{node22 } l \ a \ r'))$
by(*induct l a r' rule: node22.induct*) *auto*

lemma *complete_tree_d_node31*:
 $\llbracket \text{complete } (\text{tree}_d \ l'); \text{ complete } m; \text{ complete } r; h_d \ l' = \text{height } r; \text{ height } m = \text{height } r \rrbracket$

$\implies \text{complete } (\text{tree}_d (\text{node31 } l' a m b r))$
by(*induct l' a m b r rule: node31.induct*) *auto*

lemma *complete_tree_d_node32*:
 $\llbracket \text{complete } l; \text{complete } (\text{tree}_d m'); \text{complete } r; \text{height } l = \text{height } r; h_d m' = \text{height } r \rrbracket$
 $\implies \text{complete } (\text{tree}_d (\text{node32 } l a m' b r))$
by(*induct l a m' b r rule: node32.induct*) *auto*

lemma *complete_tree_d_node33*:
 $\llbracket \text{complete } l; \text{complete } m; \text{complete}(\text{tree}_d r'); \text{height } l = h_d r'; \text{height } m = h_d r' \rrbracket$
 $\implies \text{complete } (\text{tree}_d (\text{node33 } l a m b r'))$
by(*induct l a m b r' rule: node33.induct*) *auto*

lemmas *completes = complete_tree_d_node21 complete_tree_d_node22*
complete_tree_d_node31 complete_tree_d_node32 complete_tree_d_node33

lemma *height'_node21*:
 $\text{height } r > 0 \implies h_d(\text{node21 } l' a r) = \max (h_d l') (\text{height } r) + 1$
by(*induct l' a r rule: node21.induct*)(*simp_all*)

lemma *height'_node22*:
 $\text{height } l > 0 \implies h_d(\text{node22 } l a r') = \max (\text{height } l) (h_d r') + 1$
by(*induct l a r' rule: node22.induct*)(*simp_all*)

lemma *height'_node31*:
 $\text{height } m > 0 \implies h_d(\text{node31 } l a m b r) =$
 $\max (h_d l) (\max (\text{height } m) (\text{height } r)) + 1$
by(*induct l a m b r rule: node31.induct*)(*simp_all add: max_def*)

lemma *height'_node32*:
 $\text{height } r > 0 \implies h_d(\text{node32 } l a m b r) =$
 $\max (\text{height } l) (\max (h_d m) (\text{height } r)) + 1$
by(*induct l a m b r rule: node32.induct*)(*simp_all add: max_def*)

lemma *height'_node33*:
 $\text{height } m > 0 \implies h_d(\text{node33 } l a m b r) =$
 $\max (\text{height } l) (\max (\text{height } m) (h_d r)) + 1$
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*:
 $\text{split_min } t = (x, t') \implies \text{height } t > 0 \implies \text{complete } t \implies h_d t' = \text{height } t$
by(*induct* t *arbitrary*: $x \ t'$ *rule*: *split_min.induct*)
(auto *simp*: *heights split*: *prod.splits*)

lemma *height_del*: $\text{complete } t \implies h_d(\text{del } x \ t) = \text{height } t$
by(*induction* $x \ t$ *rule*: *del.induct*)
(auto *simp*: *heights max_def height_split_min split*: *prod.splits*)

lemma *complete_split_min*:
 $\llbracket \text{split_min } t = (x, t'); \text{complete } t; \text{height } t > 0 \rrbracket \implies \text{complete } (\text{tree}_d t')$
by(*induct* t *arbitrary*: $x \ t'$ *rule*: *split_min.induct*)
(auto *simp*: *heights height_split_min completes split*: *prod.splits*)

lemma *complete_tree_d_del*: $\text{complete } t \implies \text{complete}(\text{tree}_d(\text{del } x \ t))$
by(*induction* $x \ t$ *rule*: *del.induct*)
(auto *simp*: *completes complete_split_min height_del height_split_min split*: *prod.splits*)

corollary *complete_delete*: $\text{complete } t \implies \text{complete}(\text{delete } x \ t)$
by(*simp add*: *delete_def complete_tree_d_del*)

26.4 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

27 2-3 Tree Implementation of Maps

```

theory Tree23_Map
imports
  Tree23_Set
  Map_Specs
begin

```

```

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

```

```

fun upd :: 'a::linorder  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) tree23  $\Rightarrow$  ('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  $\Rightarrow$  (case upd x y l of
    Eqi l'  $\Rightarrow$  Eqi (Node2 l' ab r)
    | Of l1 ab' l2  $\Rightarrow$  Eqi (Node3 l1 ab' l2 ab r)) |
  EQ  $\Rightarrow$  Eqi (Node2 l (x,y) r) |
  GT  $\Rightarrow$  (case upd x y r of
    Eqi r'  $\Rightarrow$  Eqi (Node2 l ab r')
    | Of r1 ab' r2  $\Rightarrow$  Eqi (Node3 l ab r1 ab' r2))) |
upd x y (Node3 l ab1 m ab2 r) = (case cmp x (fst ab1) of
  LT  $\Rightarrow$  (case upd x y l of
    Eqi l'  $\Rightarrow$  Eqi (Node3 l' ab1 m ab2 r)
    | Of l1 ab' l2  $\Rightarrow$  Of (Node2 l1 ab' l2) ab1 (Node2 m ab2 r)) |
  EQ  $\Rightarrow$  Eqi (Node3 l (x,y) m ab2 r) |
  GT  $\Rightarrow$  (case cmp x (fst ab2) of
    LT  $\Rightarrow$  (case upd x y m of
      Eqi m'  $\Rightarrow$  Eqi (Node3 l ab1 m' ab2 r)
      | Of m1 ab' m2  $\Rightarrow$  Of (Node2 l ab1 m1) ab' (Node2 m2 ab2
r))) |
    EQ  $\Rightarrow$  Eqi (Node3 l ab1 m (x,y) r) |
    GT  $\Rightarrow$  (case upd x y r of

```

$$Eq_i \ r' \Rightarrow Eq_i \ (Node3 \ l \ ab1 \ m \ ab2 \ r')$$

$$| \ Of \ r1 \ ab' \ r2 \Rightarrow Of \ (Node2 \ l \ ab1 \ m) \ ab2 \ (Node2 \ r1 \ ab' \ r2))))$$

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

fun *del* :: 'a::linorder \Rightarrow ('a*'b) tree23 \Rightarrow ('a*'b) up_d **where**
del x Leaf = Eq_d Leaf |
del x (Node2 Leaf ab1 Leaf) = (if x=fst ab1 then Uf Leaf else Eq_d(Node2 Leaf ab1 Leaf)) |
del x (Node3 Leaf ab1 Leaf ab2 Leaf) = Eq_d(if x=fst ab1 then Node2 Leaf ab2 Leaf
 else if x=fst ab2 then Node2 Leaf ab1 Leaf else Node3 Leaf ab1 Leaf ab2 Leaf) |
del x (Node2 l ab1 r) = (case cmp x (fst ab1) of
 LT \Rightarrow node21 (del x l) ab1 r |
 GT \Rightarrow node22 l ab1 (del x r) |
 EQ \Rightarrow 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 \Rightarrow node31 (del x l) ab1 m ab2 r |
 EQ \Rightarrow let (ab1',m') = split_min m in node32 l ab1' m' ab2 r |
 GT \Rightarrow (case cmp x (fst ab2) of
 LT \Rightarrow node32 l ab1 (del x m) ab2 r |
 EQ \Rightarrow let (ab2',r') = split_min r in node33 l ab1 m ab2' r' |
 GT \Rightarrow node33 l ab1 m ab2 (del x r))))

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

27.1 Functional Correctness

lemma *lookup_map_of*:

sorted1(*inorder* t) \Longrightarrow *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*(tree_i(upd x y t)) = *upd_list* x y (*inorder* t)
by(*induction* t) (auto simp: *upd_list_simps* *split*: *up_i.splits*)

corollary *inorder_update*:

sorted1(*inorder* t) \Longrightarrow *inorder*(*update* x y t) = *upd_list* x y (*inorder* t)

by(*simp add: update_def inorder_upd*)

lemma *inorder_del*: $\llbracket \text{complete } t ; \text{sorted1}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{tree}_d(\text{del } x \ t)) = \text{del_list } x \ (\text{inorder } t)$
by(*induction t rule: del.induct*)
(auto simp: del_list_simps inorder_nodes split_minD split!: if_split prod.splits)

corollary *inorder_delete*: $\llbracket \text{complete } t ; \text{sorted1}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by(*simp add: delete_def inorder_del*)

27.2 Balancedness

lemma *complete_upd*: $\text{complete } t \implies \text{complete } (\text{tree}_i(\text{upd } x \ y \ t)) \wedge h_i(\text{upd } x \ y \ t) = \text{height } t$
by (*induct t (auto split!: if_split up_i.split)*)

corollary *complete_update*: $\text{complete } t \implies \text{complete } (\text{update } x \ y \ t)$
by (*simp add: update_def complete_upd*)

lemma *height_del*: $\text{complete } t \implies h_d(\text{del } x \ t) = \text{height } t$
by(*induction x t rule: del.induct*)
(auto simp add: heights_max_def height_split_min split: prod.split)

lemma *complete_tree_d_del*: $\text{complete } t \implies \text{complete}(\text{tree}_d(\text{del } x \ t))$
by(*induction x t rule: del.induct*)
(auto simp: completes_complete_split_min height_del height_split_min split: prod.split)

corollary *complete_delete*: $\text{complete } t \implies \text{complete}(\text{delete } x \ t)$
by(*simp add: delete_def complete_tree_d_del*)

27.3 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

```

28 2-3 Tree from List

```

theory Tree23_of_List
imports
  Tree23
  HOL-Library.Time_Commands
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.

28.1 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 ==  $\neg(\exists t. ts = T t)$ 

```

```

fun len :: 'a tree23s  $\Rightarrow$  nat where
  len (T _) = 1 |
  len (TTs _ _ ts) = len ts + 1

```

```

fun trees :: 'a tree23s  $\Rightarrow$  'a tree23 set where
  trees (T t) = {t} |
  trees (TTs t a ts) = {t}  $\cup$  trees ts

```

Join pairs of adjacent trees:

```

fun join_adj :: 'a tree23s  $\Rightarrow$  'a tree23s where
  join_adj (TTs t1 a (T t2)) = T(Node2 t1 a t2) |

```

$join_adj (TTs\ t1\ a\ (TTs\ t2\ b\ (T\ t3))) = T(Node3\ t1\ a\ t2\ b\ t3) \mid$
 $join_adj (TTs\ t1\ a\ (TTs\ t2\ b\ ts)) = TTs\ (Node2\ t1\ a\ t2)\ b\ (join_adj\ ts)$

Towards termination of *join_all*:

lemma *len_ge2*:

not_T ts $\implies len\ ts \geq 2$

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 $\implies len(join_adj\ ts) \leq len\ ts\ div\ 2$

by(*induction ts rule: join_adj.induct*) *auto*

lemma *len_join_adj1*: *not_T ts* $\implies len(join_adj\ ts) < len\ ts$

using *len_join_adj_div2[of ts] len_ge2[of ts]* **by** *simp*

corollary *len_join_adj2[termination_simp]*: $len(join_adj\ (TTs\ t\ a\ ts)) \leq len\ ts$

using *len_join_adj1[of TTs t a ts]* **by** *simp*

fun *join_all* :: '*a tree23s* \Rightarrow '*a tree23* **where**

join_all (*T t*) = *t* \mid

join_all ts = *join_all* (*join_adj ts*)

fun *leaves* :: '*a list* \Rightarrow '*a tree23s* **where**

leaves [] = *T Leaf* \mid

leaves (*a* # *as*) = *TTs Leaf a* (*leaves as*)

definition *tree23_of_list* :: '*a list* \Rightarrow '*a tree23* **where**

tree23_of_list as = *join_all*(*leaves as*)

28.2 Functional correctness

28.2.1 *inorder*:

fun *inorder2* :: '*a tree23s* \Rightarrow '*a list* **where**

inorder2 (*T t*) = *inorder t* \mid

inorder2 (*TTs t a ts*) = *inorder t* @ *a* # *inorder2 ts*

lemma *inorder2_join_adj*: *not_T ts* $\implies inorder2(join_adj\ ts) = inorder2\ ts$

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*)

28.2.2 Completeness:

lemma *complete_join_adj*:

$\forall t \in \text{trees } ts. \text{complete } t \wedge \text{height } t = n \implies \text{not_T } ts \implies$

$\forall t \in \text{trees } (\text{join_adj } ts). \text{complete } t \wedge \text{height } t = \text{Suc } n$

by (*induction ts rule: join_adj.induct*) *auto*

lemma *complete_join_all*:

$\forall t \in \text{trees } ts. \text{complete } t \wedge \text{height } t = n \implies \text{complete } (\text{join_all } ts)$

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*: $t \in \text{trees } (\text{leaves } as) \implies \text{complete } t \wedge \text{height } t =$

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*])

28.3 Linear running time

time_fun *join_adj*

```

time_fun join_all
time_fun leaves
time_fun tree23_of_list

```

```

lemma T_join_adj: not T ts  $\implies$  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 ...  $\leq$  len ?ts div 2 + 2 * (len ?ts div 2) + 1
  using len_join_adj_div2[of ?ts] by simp
  also have ...  $\leq$  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  $\leq$  3*(length as) + 3
using T_join_all[of leaves as] by(simp add: T_leaves len_leaves)

```

```

end

```

29 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

30 2-3-4 Tree Implementation of Sets

theory *Tree234_Set*

imports

Tree234

Cmp

Set_Specs

begin

declare *sorted_wrt_simps*(2)[*simp del*]

30.1 Set operations on 2-3-4 trees

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

empty = *Leaf*

fun *isin* :: '*a::linorder tree234* \Rightarrow '*a* \Rightarrow *bool* **where**

isin Leaf *x* = *False* |

isin (Node2 l a r) *x* =

(*case cmp x a of LT* \Rightarrow *isin l x* | *EQ* \Rightarrow *True* | *GT* \Rightarrow *isin r x*) |

isin (Node3 l a m b r) *x* =

(*case cmp x a of LT* \Rightarrow *isin l x* | *EQ* \Rightarrow *True* | *GT* \Rightarrow (*case cmp x b of*

LT \Rightarrow *isin m x* | *EQ* \Rightarrow *True* | *GT* \Rightarrow *isin r x*)) |

isin (Node4 t1 a t2 b t3 c t4) *x* =

(*case cmp x b of*

LT \Rightarrow

(*case cmp x a of*

LT \Rightarrow *isin t1 x* |

EQ \Rightarrow *True* |

GT \Rightarrow *isin t2 x*) |

EQ \Rightarrow *True* |

GT \Rightarrow

(*case cmp x c of*

LT \Rightarrow *isin t3 x* |

EQ \Rightarrow *True* |

GT \Rightarrow *isin t4 x*))

datatype '*a up_i* = *T_i* '*a tree234* | *Up_i* '*a tree234* '*a* '*a tree234*

fun *tree_i* :: '*a up_i* \Rightarrow '*a tree234* **where**

tree_i (*T_i* *t*) = *t* |

$tree_i (Up_i l a r) = Node2 l a r$

fun $ins :: 'a::linorder \Rightarrow 'a tree234 \Rightarrow 'a up_i$ **where**

$ins\ x\ Leaf = Up_i\ Leaf\ x\ Leaf\ |$

$ins\ x\ (Node2\ l\ a\ r) =$

$(case\ cmp\ x\ a\ of$

$LT \Rightarrow (case\ ins\ x\ l\ of$

$T_i\ l' \Rightarrow T_i\ (Node2\ l'\ a\ r)$

$| Up_i\ l1\ b\ l2 \Rightarrow T_i\ (Node3\ l1\ b\ l2\ a\ r))\ |$

$EQ \Rightarrow T_i\ (Node2\ l\ x\ r)\ |$

$GT \Rightarrow (case\ ins\ x\ r\ of$

$T_i\ r' \Rightarrow T_i\ (Node2\ l\ a\ r')$

$| Up_i\ r1\ b\ r2 \Rightarrow T_i\ (Node3\ l\ a\ r1\ b\ r2)))\ |$

$ins\ x\ (Node3\ l\ a\ m\ b\ r) =$

$(case\ cmp\ x\ a\ of$

$LT \Rightarrow (case\ ins\ x\ l\ of$

$T_i\ l' \Rightarrow T_i\ (Node3\ l'\ a\ m\ b\ r)$

$| Up_i\ l1\ c\ l2 \Rightarrow Up_i\ (Node2\ l1\ c\ l2)\ a\ (Node2\ m\ b\ r))\ |$

$EQ \Rightarrow T_i\ (Node3\ l\ a\ m\ b\ r)\ |$

$GT \Rightarrow (case\ cmp\ x\ b\ of$

$GT \Rightarrow (case\ ins\ x\ r\ of$

$T_i\ r' \Rightarrow T_i\ (Node3\ l\ a\ m\ b\ r')$

$| Up_i\ r1\ c\ r2 \Rightarrow Up_i\ (Node2\ l\ a\ m)\ b\ (Node2\ r1\ c\ r2))\ |$

$EQ \Rightarrow T_i\ (Node3\ l\ a\ m\ b\ r)\ |$

$LT \Rightarrow (case\ ins\ x\ m\ of$

$T_i\ m' \Rightarrow T_i\ (Node3\ l\ a\ m'\ b\ r)$

$| Up_i\ m1\ c\ m2 \Rightarrow Up_i\ (Node2\ l\ a\ m1)\ c\ (Node2\ m2\ b$

$r))))\ |$

$ins\ x\ (Node4\ t1\ a\ t2\ b\ t3\ c\ t4) =$

$(case\ cmp\ x\ b\ of$

$LT \Rightarrow$

$(case\ cmp\ x\ a\ of$

$LT \Rightarrow$

$(case\ ins\ x\ t1\ of$

$T_i\ t \Rightarrow T_i\ (Node4\ t\ a\ t2\ b\ t3\ c\ t4)\ |$

$Up_i\ l\ y\ r \Rightarrow Up_i\ (Node2\ l\ y\ r)\ a\ (Node3\ t2\ b\ t3\ c\ t4))\ |$

$EQ \Rightarrow T_i\ (Node4\ t1\ a\ t2\ b\ t3\ c\ t4)\ |$

$GT \Rightarrow$

$(case\ ins\ x\ t2\ of$

$T_i\ t \Rightarrow T_i\ (Node4\ t1\ a\ t\ b\ t3\ c\ t4)\ |$

$Up_i\ l\ y\ r \Rightarrow Up_i\ (Node2\ t1\ a\ l)\ y\ (Node3\ r\ b\ t3\ c\ t4)))\ |$

$EQ \Rightarrow T_i\ (Node4\ t1\ a\ t2\ b\ t3\ c\ t4)\ |$

$GT \Rightarrow$

$(case\ cmp\ x\ c\ of$

```

LT ⇒
  (case ins x t3 of
    Ti t => Ti (Node4 t1 a t2 b t c t4) |
    Upi l y r => Upi (Node2 t1 a t2) b (Node3 l y r c t4)) |
EQ ⇒ Ti (Node4 t1 a t2 b t3 c t4) |
GT ⇒
  (case ins x t4 of
    Ti t => Ti (Node4 t1 a t2 b t3 c t) |
    Upi l y r => Upi (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 $node32 :: 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ up_d \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \ up_d$ **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 $node33 :: 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ up_d \Rightarrow 'a \ up_d$ **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 $node41 :: 'a \ up_d \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \ up_d$ **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 $node42 :: 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ up_d \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \ up_d$ **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 $node43 :: 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \Rightarrow 'a \ up_d \Rightarrow 'a \Rightarrow 'a \ tree234 \Rightarrow 'a \ up_d$ **where**
 $node43 \ t1 \ a \ t2 \ b \ (T_d \ t3) \ c \ t4 = T_d(Node4 \ t1 \ a \ t2 \ b \ t3 \ c \ t4) \ |$
 $node43 \ t1 \ a \ (Node2 \ t2 \ b \ t3) \ c \ (Up_d \ t4) \ d \ t5 = T_d(Node3 \ t1 \ a \ (Node3 \ t2 \ b \ t3) \ c \ t4 \ d \ t5)$

$t3\ c\ t4)\ d\ t5) \mid$
 $node43\ t1\ a\ (Node3\ t2\ b\ t3\ c\ t4)\ d\ (Up_d\ t5)\ e\ t6 = T_d(Node4\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node2\ t4\ d\ t5)\ e\ t6) \mid$
 $node43\ t1\ a\ (Node4\ t2\ b\ t3\ c\ t4\ d\ t5)\ e\ (Up_d\ t6)\ f\ t7 = T_d(Node4\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node3\ t4\ d\ t5\ e\ t6)\ f\ t7)$

fun $node44 :: 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a$
 $up_d \Rightarrow 'a\ up_d$ **where**
 $node44\ t1\ a\ t2\ b\ t3\ c\ (T_d\ t4) = T_d(Node4\ t1\ a\ t2\ b\ t3\ c\ t4) \mid$
 $node44\ t1\ a\ t2\ b\ (Node2\ t3\ c\ t4)\ d\ (Up_d\ t5) = T_d(Node3\ t1\ a\ t2\ b\ (Node3\ t3\ c\ t4\ d\ t5)) \mid$
 $node44\ t1\ a\ t2\ b\ (Node3\ t3\ c\ t4\ d\ t5)\ e\ (Up_d\ t6) = T_d(Node4\ t1\ a\ t2\ b\ (Node2\ t3\ c\ t4)\ d\ (Node2\ t5\ e\ t6)) \mid$
 $node44\ t1\ a\ t2\ b\ (Node4\ t3\ c\ t4\ d\ t5\ e\ t6)\ f\ (Up_d\ t7) = T_d(Node4\ t1\ a\ t2\ b\ (Node2\ t3\ c\ t4)\ d\ (Node3\ t5\ e\ t6\ f\ t7))$

fun $split_min :: 'a\ tree234 \Rightarrow 'a * 'a\ up_d$ **where**
 $split_min\ (Node2\ Leaf\ a\ Leaf) = (a,\ Up_d\ Leaf) \mid$
 $split_min\ (Node3\ Leaf\ a\ Leaf\ b\ Leaf) = (a,\ T_d(Node2\ Leaf\ b\ Leaf)) \mid$
 $split_min\ (Node4\ Leaf\ a\ Leaf\ b\ Leaf\ c\ Leaf) = (a,\ T_d(Node3\ Leaf\ b\ Leaf\ c\ Leaf)) \mid$
 $split_min\ (Node2\ l\ a\ r) = (let\ (x,l') = split_min\ l\ in\ (x,\ node21\ l'\ a\ r)) \mid$
 $split_min\ (Node3\ l\ a\ m\ b\ r) = (let\ (x,l') = split_min\ l\ in\ (x,\ node31\ l'\ a\ m\ b\ r)) \mid$
 $split_min\ (Node4\ l\ a\ m\ b\ n\ c\ r) = (let\ (x,l') = split_min\ l\ in\ (x,\ node41\ l'\ a\ m\ b\ n\ c\ r))$

fun $del :: 'a::linorder \Rightarrow 'a\ tree234 \Rightarrow 'a\ up_d$ **where**
 $del\ k\ Leaf = T_d\ Leaf \mid$
 $del\ k\ (Node2\ Leaf\ p\ Leaf) = (if\ k=p\ then\ Up_d\ Leaf\ else\ T_d(Node2\ Leaf\ p\ Leaf)) \mid$
 $del\ k\ (Node3\ Leaf\ p\ Leaf\ q\ Leaf) = T_d(if\ k=p\ then\ Node2\ Leaf\ q\ Leaf\ else\ if\ k=q\ then\ Node2\ Leaf\ p\ Leaf\ else\ Node3\ Leaf\ p\ Leaf\ q\ Leaf) \mid$
 $del\ k\ (Node4\ Leaf\ a\ Leaf\ b\ Leaf\ c\ Leaf) =$
 $\quad T_d(if\ k=a\ then\ Node3\ Leaf\ b\ Leaf\ c\ Leaf\ else$
 $\quad\quad if\ k=b\ then\ Node3\ Leaf\ a\ Leaf\ c\ Leaf\ else$
 $\quad\quad if\ k=c\ then\ Node3\ Leaf\ a\ Leaf\ b\ Leaf$
 $\quad\quad\quad else\ Node4\ Leaf\ a\ Leaf\ b\ Leaf\ c\ Leaf) \mid$
 $del\ k\ (Node2\ l\ a\ r) = (case\ cmp\ k\ a\ of$
 $\quad LT \Rightarrow node21\ (del\ k\ l)\ a\ r \mid$
 $\quad GT \Rightarrow node22\ l\ a\ (del\ k\ r) \mid$
 $\quad EQ \Rightarrow let\ (a',t) = split_min\ r\ in\ node22\ l\ a'\ t) \mid$
 $del\ k\ (Node3\ l\ a\ m\ b\ r) = (case\ cmp\ k\ a\ of$
 $\quad LT \Rightarrow node31\ (del\ k\ l)\ a\ m\ b\ r \mid$

$EQ \Rightarrow \text{let } (a', m') = \text{split_min } m \text{ in node32 } l \ a' \ m' \ b \ r \mid$
 $GT \Rightarrow (\text{case cmp } k \ b \text{ of}$
 $\quad LT \Rightarrow \text{node32 } l \ a \ (\text{del } k \ m) \ b \ r \mid$
 $\quad EQ \Rightarrow \text{let } (b', r') = \text{split_min } r \text{ in node33 } l \ a \ m \ b' \ r' \mid$
 $\quad GT \Rightarrow \text{node33 } l \ a \ m \ b \ (\text{del } k \ r))) \mid$
 $\text{del } k \ (\text{Node4 } l \ a \ m \ b \ n \ c \ r) = (\text{case cmp } k \ b \text{ of}$
 $\quad LT \Rightarrow (\text{case cmp } k \ a \text{ of}$
 $\quad\quad LT \Rightarrow \text{node41 } (\text{del } k \ l) \ a \ m \ b \ n \ c \ r \mid$
 $\quad\quad EQ \Rightarrow \text{let } (a', m') = \text{split_min } m \text{ in node42 } l \ a' \ m' \ b \ n \ c \ r \mid$
 $\quad\quad GT \Rightarrow \text{node42 } l \ a \ (\text{del } k \ m) \ b \ n \ c \ r) \mid$
 $\quad EQ \Rightarrow \text{let } (b', n') = \text{split_min } n \text{ in node43 } l \ a \ m \ b' \ n' \ c \ r \mid$
 $\quad GT \Rightarrow (\text{case cmp } k \ c \text{ of}$
 $\quad\quad LT \Rightarrow \text{node43 } l \ a \ m \ b \ (\text{del } k \ n) \ c \ r \mid$
 $\quad\quad EQ \Rightarrow \text{let } (c', r') = \text{split_min } r \text{ in node44 } l \ a \ m \ b \ n \ c' \ r' \mid$
 $\quad\quad GT \Rightarrow \text{node44 } l \ a \ m \ b \ n \ c \ (\text{del } k \ r)))$

definition $\text{delete} :: 'a :: \text{linorder} \Rightarrow 'a \text{ tree234} \Rightarrow 'a \text{ tree234}$ **where**
 $\text{delete } x \ t = \text{tree}_d(\text{del } x \ t)$

30.2 Functional correctness

30.2.1 Functional correctness of isin:

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

30.2.2 Functional correctness of insert:

lemma $\text{inorder_ins}: \text{sorted}(\text{inorder } t) \Longrightarrow \text{inorder}(\text{tree}_i(\text{ins } x \ t)) = \text{ins_list } x \ (\text{inorder } t)$
by $(\text{induction } t) (\text{auto}, \text{auto simp: ins_list_simps split!: if_splits up_i.splits})$

lemma $\text{inorder_insert}: \text{sorted}(\text{inorder } t) \Longrightarrow \text{inorder}(\text{insert } a \ t) = \text{ins_list } a \ (\text{inorder } t)$
by $(\text{simp add: insert_def inorder_ins})$

30.2.3 Functional correctness of delete

lemma $\text{inorder_node21}: \text{height } r > 0 \Longrightarrow \text{inorder } (\text{tree}_d(\text{node21 } l' \ a \ r)) = \text{inorder } (\text{tree}_d \ l') @ a \ \# \text{inorder } r$
by $(\text{induct } l' \ a \ r \text{ rule: node21.induct}) \text{ auto}$

lemma $\text{inorder_node22}: \text{height } l > 0 \Longrightarrow \text{inorder } (\text{tree}_d(\text{node22 } l \ a \ r')) = \text{inorder } l @ a \ \# \text{inorder } (\text{tree}_d \ r')$
by $(\text{induct } l \ a \ r' \text{ rule: node22.induct}) \text{ auto}$

lemma *inorder_node31*: $\text{height } m > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node31 } l' a m b r)) = \text{inorder } (\text{tree}_d l') @ a \# \text{inorder } m$
 $@ b \# \text{inorder } r$
by(*induct* $l' a m b r$ *rule*: *node31.induct*) *auto*

lemma *inorder_node32*: $\text{height } r > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node32 } l a m' b r)) = \text{inorder } l @ a \# \text{inorder } (\text{tree}_d m')$
 $@ b \# \text{inorder } r$
by(*induct* $l a m' b r$ *rule*: *node32.induct*) *auto*

lemma *inorder_node33*: $\text{height } m > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node33 } l a m b r')) = \text{inorder } l @ a \# \text{inorder } m @ b \#$
 $\text{inorder } (\text{tree}_d r')$
by(*induct* $l a m b r'$ *rule*: *node33.induct*) *auto*

lemma *inorder_node41*: $\text{height } m > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node41 } l' a m b n c r)) = \text{inorder } (\text{tree}_d l') @ a \# \text{inorder } m$
 $@ b \# \text{inorder } n @ c \# \text{inorder } r$
by(*induct* $l' a m b n c r$ *rule*: *node41.induct*) *auto*

lemma *inorder_node42*: $\text{height } l > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node42 } l a m b n c r)) = \text{inorder } l @ a \# \text{inorder } (\text{tree}_d m)$
 $@ b \# \text{inorder } n @ c \# \text{inorder } r$
by(*induct* $l a m b n c r$ *rule*: *node42.induct*) *auto*

lemma *inorder_node43*: $\text{height } m > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node43 } l a m b n c r)) = \text{inorder } l @ a \# \text{inorder } m @ b$
 $\# \text{inorder } (\text{tree}_d n) @ c \# \text{inorder } r$
by(*induct* $l a m b n c r$ *rule*: *node43.induct*) *auto*

lemma *inorder_node44*: $\text{height } n > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node44 } l a m b n c r)) = \text{inorder } l @ a \# \text{inorder } m @ b$
 $\# \text{inorder } n @ c \# \text{inorder } (\text{tree}_d r)$
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*:
 $\text{split_min } t = (x, t') \implies \text{bal } t \implies \text{height } t > 0 \implies$
 $x \# \text{inorder } (\text{tree}_d t') = \text{inorder } t$
by(*induction* t *arbitrary*: t' *rule*: *split_min.induct*)

(auto simp: inorder_nodes split: prod.splits)

lemma *inorder_del*: $\llbracket \text{bal } t ; \text{sorted}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{tree}_d(\text{del } x \ t)) = \text{del_list } x \ (\text{inorder } t)$
by(*induction* *t* *rule*: *del.induct*)
(auto simp: inorder_nodes del_list_simps split_minD split!: if_split prod.splits)

lemma *inorder_delete*: $\llbracket \text{bal } t ; \text{sorted}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by(*simp* *add*: *delete_def* *inorder_del*)

30.3 Balancedness

30.3.1 Proofs for insert

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

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

fun *height_up_i* :: '*a up_i* \Rightarrow nat **where**
 $\text{height } (T_i \ t) = \text{height } t \mid$
 $\text{height } (Up_i \ l \ a \ r) = \text{height } l$

instance ..

end

lemma *bal_ins*: $\text{bal } t \implies \text{bal } (\text{tree}_i(\text{ins } a \ t)) \wedge \text{height}(\text{ins } a \ t) = \text{height } t$
by (*induct* *t*) (auto *split*!: *if_split* *up_i.split*)

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

inductive *full* :: nat \Rightarrow '*a tree234* \Rightarrow bool **where**
 $\text{full } 0 \ \text{Leaf} \mid$
 $\llbracket \text{full } n \ l ; \text{full } n \ r \rrbracket \implies \text{full } (\text{Suc } n) \ (\text{Node2 } l \ p \ r) \mid$
 $\llbracket \text{full } n \ l ; \text{full } n \ m ; \text{full } n \ r \rrbracket \implies \text{full } (\text{Suc } n) \ (\text{Node3 } l \ p \ m \ q \ r) \mid$
 $\llbracket \text{full } n \ l ; \text{full } n \ m ; \text{full } n \ m' ; \text{full } n \ r \rrbracket \implies \text{full } (\text{Suc } n) \ (\text{Node4 } l \ p \ m \ q \ m' \ q' \ r)$

inductive_cases *full_elim*s:
 $\text{full } n \ \text{Leaf}$
 $\text{full } n \ (\text{Node2 } l \ p \ r)$
 $\text{full } n \ (\text{Node3 } l \ p \ m \ q \ r)$

$full\ n\ (Node4\ l\ p\ m\ q\ m'\ q'\ r)$

inductive_cases $full_0_elim$: $full\ 0\ t$

inductive_cases $full_Suc_elim$: $full\ (Suc\ n)\ t$

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

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

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

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

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

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

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

lemma bal_imp_full : $bal\ t \implies full\ (height\ t)\ t$
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*: $\text{full } n \ t \implies \text{full}_i \ n \ (\text{ins } a \ t)$
by (*induct rule: full.induct*) (*auto, auto split: up_i.split*)

The *insert* operation preserves balance.

lemma *bal_insert*: $\text{bal } t \implies \text{bal } (\text{insert } a \ t)$
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

30.3.2 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*:
 $\llbracket \text{bal } r; \text{bal } (\text{tree}_d \ l); \text{height } r = \text{height } l \rrbracket \implies \text{bal } (\text{tree}_d \ (\text{node21 } l \ a \ r))$
by(*induct l a r rule: node21.induct*) *auto*

lemma *bal_tree_d_node22*:
 $\llbracket \text{bal}(\text{tree}_d \ r); \text{bal } l; \text{height } r = \text{height } l \rrbracket \implies \text{bal } (\text{tree}_d \ (\text{node22 } l \ a \ r))$
by(*induct l a r rule: node22.induct*) *auto*

lemma *bal_tree_d_node31*:
 $\llbracket \text{bal } (\text{tree}_d \ l); \text{bal } m; \text{bal } r; \text{height } l = \text{height } r; \text{height } m = \text{height } r \rrbracket$
 $\implies \text{bal } (\text{tree}_d \ (\text{node31 } l \ a \ m \ b \ r))$
by(*induct l a m b r rule: node31.induct*) *auto*

lemma *bal_tree_d_node32*:
 $\llbracket \text{bal } l; \text{bal } (\text{tree}_d \ m); \text{bal } r; \text{height } l = \text{height } r; \text{height } m = \text{height } r \rrbracket$
 $\implies \text{bal } (\text{tree}_d \ (\text{node32 } l \ a \ m \ b \ r))$
by(*induct l a m b r rule: node32.induct*) *auto*

lemma *bal_tree_d_node33*:

$\llbracket \text{bal } l; \text{bal } m; \text{bal}(\text{tree}_d r); \text{height } l = \text{height } r; \text{height } m = \text{height } r \rrbracket$
 $\implies \text{bal } (\text{tree}_d (\text{node33 } l a m b r))$
by(*induct l a m b r rule: node33.induct*) *auto*

lemma *bal_tree_d_node41*:
 $\llbracket \text{bal } (\text{tree}_d l); \text{bal } m; \text{bal } n; \text{bal } r; \text{height } l = \text{height } r; \text{height } m = \text{height } r; \text{height } n = \text{height } r \rrbracket$
 $\implies \text{bal } (\text{tree}_d (\text{node41 } l a m b n c r))$
by(*induct l a m b n c r rule: node41.induct*) *auto*

lemma *bal_tree_d_node42*:
 $\llbracket \text{bal } l; \text{bal } (\text{tree}_d m); \text{bal } n; \text{bal } r; \text{height } l = \text{height } r; \text{height } m = \text{height } r; \text{height } n = \text{height } r \rrbracket$
 $\implies \text{bal } (\text{tree}_d (\text{node42 } l a m b n c r))$
by(*induct l a m b n c r rule: node42.induct*) *auto*

lemma *bal_tree_d_node43*:
 $\llbracket \text{bal } l; \text{bal } m; \text{bal } (\text{tree}_d n); \text{bal } r; \text{height } l = \text{height } r; \text{height } m = \text{height } r; \text{height } n = \text{height } r \rrbracket$
 $\implies \text{bal } (\text{tree}_d (\text{node43 } l a m b n c r))$
by(*induct l a m b n c r rule: node43.induct*) *auto*

lemma *bal_tree_d_node44*:
 $\llbracket \text{bal } l; \text{bal } m; \text{bal } n; \text{bal } (\text{tree}_d r); \text{height } l = \text{height } r; \text{height } m = \text{height } r; \text{height } n = \text{height } r \rrbracket$
 $\implies \text{bal } (\text{tree}_d (\text{node44 } l a m b n c r))$
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*:
 $\text{height } r > 0 \implies \text{height}(\text{node21 } l a r) = \max (\text{height } l) (\text{height } r) + 1$
by(*induct l a r rule: node21.induct*)(*simp_all add: max.assoc*)

lemma *height_node22*:
 $\text{height } l > 0 \implies \text{height}(\text{node22 } l a r) = \max (\text{height } l) (\text{height } r) + 1$
by(*induct l a r rule: node22.induct*)(*simp_all add: max.assoc*)

lemma *height_node31*:
 $\text{height } m > 0 \implies \text{height}(\text{node31 } l a m b r) =$
 $\max (\text{height } l) (\max (\text{height } m) (\text{height } r)) + 1$
by(*induct l a m b r rule: node31.induct*)(*simp_all add: max_def*)

lemma *height_node32*:

$\text{height } r > 0 \implies \text{height}(\text{node32 } l \ a \ m \ b \ r) =$
 $\max (\text{height } l) (\max (\text{height } m) (\text{height } r)) + 1$

by(*induct* *l a m b r* *rule*: *node32.induct*)(*simp_all* *add*: *max_def*)

lemma *height_node33*:

$\text{height } m > 0 \implies \text{height}(\text{node33 } l \ a \ m \ b \ r) =$
 $\max (\text{height } l) (\max (\text{height } m) (\text{height } r)) + 1$

by(*induct* *l a m b r* *rule*: *node33.induct*)(*simp_all* *add*: *max_def*)

lemma *height_node41*:

$\text{height } m > 0 \implies \text{height}(\text{node41 } l \ a \ m \ b \ n \ c \ r) =$
 $\max (\text{height } l) (\max (\text{height } m) (\max (\text{height } n) (\text{height } r))) + 1$

by(*induct* *l a m b n c r* *rule*: *node41.induct*)(*simp_all* *add*: *max_def*)

lemma *height_node42*:

$\text{height } l > 0 \implies \text{height}(\text{node42 } l \ a \ m \ b \ n \ c \ r) =$
 $\max (\text{height } l) (\max (\text{height } m) (\max (\text{height } n) (\text{height } r))) + 1$

by(*induct* *l a m b n c r* *rule*: *node42.induct*)(*simp_all* *add*: *max_def*)

lemma *height_node43*:

$\text{height } m > 0 \implies \text{height}(\text{node43 } l \ a \ m \ b \ n \ c \ r) =$
 $\max (\text{height } l) (\max (\text{height } m) (\max (\text{height } n) (\text{height } r))) + 1$

by(*induct* *l a m b n c r* *rule*: *node43.induct*)(*simp_all* *add*: *max_def*)

lemma *height_node44*:

$\text{height } n > 0 \implies \text{height}(\text{node44 } l \ a \ m \ b \ n \ c \ r) =$
 $\max (\text{height } l) (\max (\text{height } m) (\max (\text{height } n) (\text{height } r))) + 1$

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*:

$\text{split_min } t = (x, t') \implies \text{height } t > 0 \implies \text{bal } t \implies \text{height } t' = \text{height } t$

by(*induct* *t* *arbitrary*: *x t'* *rule*: *split_min.induct*)

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

lemma *height_del*: $\text{bal } t \implies \text{height}(\text{del } x \ t) = \text{height } t$

by(*induction* *x t* *rule*: *del.induct*)

(*auto simp* *add*: *heights height_split_min split*!: *if_split prod.split*)

```

lemma bal_split_min:
   $\llbracket \text{split\_min } t = (x, t'); \text{ bal } t; \text{ height } t > 0 \rrbracket \implies \text{ bal } (\text{tree}_d t)$ 
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  $\implies$  bal(treed(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  $\implies$  bal(delete x t)
by(simp add: delete_def bal_tree_d_del)

```

30.4 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

```

31 2-3-4 Tree Implementation of Maps

```

theory Tree234_Map
imports
  Tree234_Set
  Map_Specs
begin

```

31.1 Map operations on 2-3-4 trees

```

fun lookup :: ('a::linorder * 'b) tree234  $\Rightarrow$  'a  $\Rightarrow$  'b option where

```

```

lookup Leaf x = None |
lookup (Node2 l (a,b) r) x = (case cmp x a of
  LT => 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)))) |

```

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

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

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

$del\ x\ (Node3\ l\ ab1\ m\ ab2\ r) = (case\ cmp\ x\ (fst\ ab1)\ of$
 $LT \Rightarrow node31\ (del\ x\ l)\ ab1\ m\ ab2\ r\ |$
 $EQ \Rightarrow let\ (ab1',m') = split_min\ m\ in\ node32\ l\ ab1'\ m'\ ab2\ r\ |$
 $GT \Rightarrow (case\ cmp\ x\ (fst\ ab2)\ of$
 $LT \Rightarrow node32\ l\ ab1\ (del\ x\ m)\ ab2\ r\ |$
 $EQ \Rightarrow let\ (ab2',r') = split_min\ r\ in\ node33\ l\ ab1\ m\ ab2'\ r'\ |$
 $GT \Rightarrow node33\ l\ ab1\ m\ ab2\ (del\ x\ r)))\ |$
 $del\ x\ (Node4\ t1\ ab1\ t2\ ab2\ t3\ ab3\ t4) = (case\ cmp\ x\ (fst\ ab2)\ of$
 $LT \Rightarrow (case\ cmp\ x\ (fst\ ab1)\ of$
 $LT \Rightarrow node41\ (del\ x\ t1)\ ab1\ t2\ ab2\ t3\ ab3\ t4\ |$
 $EQ \Rightarrow let\ (ab',t2') = split_min\ t2\ in\ node42\ t1\ ab'\ t2'\ ab2\ t3\ ab3$
 $t4\ |$
 $GT \Rightarrow node42\ t1\ ab1\ (del\ x\ t2)\ ab2\ t3\ ab3\ t4)\ |$
 $EQ \Rightarrow let\ (ab',t3') = split_min\ t3\ in\ node43\ t1\ ab1\ t2\ ab'\ t3'\ ab3\ t4\ |$
 $GT \Rightarrow (case\ cmp\ x\ (fst\ ab3)\ of$
 $LT \Rightarrow node43\ t1\ ab1\ t2\ ab2\ (del\ x\ t3)\ ab3\ t4\ |$
 $EQ \Rightarrow let\ (ab',t4') = split_min\ t4\ in\ node44\ t1\ ab1\ t2\ ab2\ t3\ ab'$
 $t4'\ |$
 $GT \Rightarrow node44\ t1\ ab1\ t2\ ab2\ t3\ ab3\ (del\ x\ t4)))$

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

31.2 Functional correctness

lemma $lookup_map_of$:

$sorted1(inorder\ t) \implies lookup\ t\ x = map_of\ (inorder\ t)\ x$
by ($induction\ t$) ($auto\ simp: map_of_simps\ split: option.split$)

lemma $inorder_upd$:

$sorted1(inorder\ t) \implies inorder(tree_i(upd\ a\ b\ t)) = upd_list\ a\ b\ (inorder\ t)$
by($induction\ t$)
 $(auto\ simp: upd_list_simps, auto\ simp: upd_list_simps\ split: up_i.splits)$

lemma $inorder_update$:

$sorted1(inorder\ t) \implies inorder(update\ a\ b\ t) = upd_list\ a\ b\ (inorder\ t)$
by($simp\ add: update_def\ inorder_upd$)

lemma $inorder_del$: $\llbracket bal\ t ; sorted1(inorder\ t) \rrbracket \implies$

$inorder(tree_d\ (del\ x\ t)) = del_list\ x\ (inorder\ t)$

by($induction\ t\ rule: del.induct$)

$(auto\ simp: del_list_simps\ inorder_nodes\ split_minD\ split!: if_splits\ prod.splits)$

lemma *inorder_delete*: $\llbracket \text{bal } t ; \text{sorted1}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by (*simp add: delete_def inorder_del*)

31.3 Balancedness

lemma *bal_upd*: $\text{bal } t \implies \text{bal } (\text{tree}_i(\text{upd } x \ y \ t)) \wedge \text{height}(\text{upd } x \ y \ t) = \text{height } t$
by (*induct t*) (*auto, auto split!: if_split up_i.split*)

lemma *bal_update*: $\text{bal } t \implies \text{bal } (\text{update } x \ y \ t)$
by (*simp add: update_def bal_upd*)

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

lemma *bal_tree_d_del*: $\text{bal } t \implies \text{bal}(\text{tree}_d(\text{del } x \ t))$
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*: $\text{bal } t \implies \text{bal}(\text{delete } x \ t)$
by (*simp add: delete_def bal_tree_d_del*)

31.4 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

32 1-2 Brother Tree Implementation of Sets

```
theory Brother12_Set
  imports
    Cmp
    Set_Specs
    HOL-Number_Theory.Fib
begin
```

32.1 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  $\Rightarrow$  '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  $\Rightarrow$  'a::linorder  $\Rightarrow$  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  $\Rightarrow$  isin l x |
      EQ  $\Rightarrow$  True |
      GT  $\Rightarrow$  isin r x)
```

```
fun n1 :: 'a bro  $\Rightarrow$  '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 $n2 :: 'a\ bro \Rightarrow 'a \Rightarrow 'a\ bro \Rightarrow 'a\ bro$ **where**
 $n2\ (L2\ a1)\ a2\ t = N3\ N0\ a1\ N0\ a2\ t \mid$
 $n2\ (N3\ t1\ a1\ t2\ a2\ t3)\ a3\ (N1\ t4) = N2\ (N2\ t1\ a1\ t2)\ a2\ (N2\ t3\ a3\ t4) \mid$
 $n2\ (N3\ t1\ a1\ t2\ a2\ t3)\ a3\ t4 = N3\ (N2\ t1\ a1\ t2)\ a2\ (N1\ t3)\ a3\ t4 \mid$
 $n2\ t1\ a1\ (L2\ a2) = N3\ t1\ a1\ N0\ a2\ N0 \mid$
 $n2\ (N1\ t1)\ a1\ (N3\ t2\ a2\ t3\ a3\ t4) = N2\ (N2\ t1\ a1\ t2)\ a2\ (N2\ t3\ a3\ t4) \mid$
 $n2\ t1\ a1\ (N3\ t2\ a2\ t3\ a3\ t4) = N3\ t1\ a1\ (N1\ t2)\ a2\ (N2\ t3\ a3\ t4) \mid$
 $n2\ t1\ a\ t2 = N2\ t1\ a\ t2$

fun $ins :: 'a::linorder \Rightarrow 'a\ bro \Rightarrow 'a\ bro$ **where**

$ins\ x\ N0 = L2\ x \mid$
 $ins\ x\ (N1\ t) = n1\ (ins\ x\ t) \mid$
 $ins\ x\ (N2\ l\ a\ r) =$
 $(case\ cmp\ x\ a\ of$
 $\quad LT \Rightarrow n2\ (ins\ x\ l)\ a\ r \mid$
 $\quad EQ \Rightarrow N2\ l\ a\ r \mid$
 $\quad GT \Rightarrow n2\ l\ a\ (ins\ x\ r))$

fun $tree :: 'a\ bro \Rightarrow 'a\ bro$ **where**

$tree\ (L2\ a) = N2\ N0\ a\ N0 \mid$
 $tree\ (N3\ t1\ a1\ t2\ a2\ t3) = N2\ (N2\ t1\ a1\ t2)\ a2\ (N1\ t3) \mid$
 $tree\ t = t$

definition $insert :: 'a::linorder \Rightarrow 'a\ bro \Rightarrow 'a\ bro$ **where**

$insert\ x\ t = tree(ins\ x\ t)$

end

locale *delete*

begin

fun $n2 :: 'a\ bro \Rightarrow 'a \Rightarrow 'a\ bro \Rightarrow 'a\ bro$ **where**

$n2\ (N1\ t1)\ a1\ (N1\ t2) = N1\ (N2\ t1\ a1\ t2) \mid$
 $n2\ (N1\ (N1\ t1))\ a1\ (N2\ (N1\ t2)\ a2\ (N2\ t3\ a3\ t4)) =$
 $N1\ (N2\ (N2\ t1\ a1\ t2)\ a2\ (N2\ t3\ a3\ t4)) \mid$
 $n2\ (N1\ (N1\ t1))\ a1\ (N2\ (N2\ t2\ a2\ t3)\ a3\ (N1\ t4)) =$
 $N1\ (N2\ (N2\ t1\ a1\ t2)\ a2\ (N2\ t3\ a3\ t4)) \mid$

```

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  $\Rightarrow$  ('a  $\times$  'a bro) option where
  split_min N0 = None |
  split_min (N1 t) =
    (case split_min t of
      None  $\Rightarrow$  None |
      Some (a, t')  $\Rightarrow$  Some (a, N1 t')) |
  split_min (N2 t1 a t2) =
    (case split_min t1 of
      None  $\Rightarrow$  Some (a, N1 t2) |
      Some (b, t1')  $\Rightarrow$  Some (b, n2 t1' a t2))

```

```

fun del :: 'a::linorder  $\Rightarrow$  'a bro  $\Rightarrow$  '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  $\Rightarrow$  n2 (del x l) a r |
      GT  $\Rightarrow$  n2 l a (del x r) |
      EQ  $\Rightarrow$  (case split_min r of
        None  $\Rightarrow$  N1 l |
        Some (b, r')  $\Rightarrow$  n2 l b r'))

```

```

fun tree :: 'a bro  $\Rightarrow$  'a bro where
  tree (N1 t) = t |
  tree t = t

```

```

definition delete :: 'a::linorder  $\Rightarrow$  'a bro  $\Rightarrow$  'a bro where
  delete a t = tree (del a t)

```

end

32.2 Invariants

```

fun B :: nat  $\Rightarrow$  'a bro set

```

and $U :: \text{nat} \Rightarrow 'a \text{ bro set}$ **where**
 $B \ 0 = \{N0\} \mid$
 $B \ (Suc \ h) = \{ \ N2 \ t1 \ a \ t2 \mid t1 \ a \ t2. \ t1 \in B \ h \cup U \ h \wedge t2 \in B \ h \vee t1 \in B \ h \wedge t2 \in B \ h \cup U \ h \} \mid$
 $U \ 0 = \{\} \mid$
 $U \ (Suc \ h) = N1 \ ' \ B \ h$

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

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

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

32.3 Functional Correctness Proofs

32.3.1 Proofs for isin

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

32.3.2 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 $inorder_ins$: $t \in T \ h \implies$
 $sorted(inorder \ t) \implies inorder(ins \ a \ t) = ins_list \ a \ (inorder \ t)$

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

lemma *inorder_insert*: $t \in T \implies$
 $\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{insert } a \ t) = \text{ins_list } a \ (\text{inorder } t)$
by(*simp add: insert_def inorder_ins inorder_tree*)

end

32.3.3 Proofs for deletion

context *delete*
begin

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

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

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

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

lemma *inorder_delete*:
 $t \in T \implies \text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by(*simp add: delete_def inorder_del inorder_tree*)

end

32.4 Invariant Proofs

32.4.1 Proofs for insertion

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

context *insert*
begin

lemma *tree_type*: $t \in Bp\ h \implies tree\ t \in B\ h \cup B\ (Suc\ h)$
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*: $t \in B\ h \implies t \in Bp\ h$
by (*cases h rule: Bp.cases*) *simp_all*

An automatic proof:

lemma
 $(t \in B\ h \longrightarrow ins\ x\ t \in Bp\ h) \wedge (t \in U\ h \longrightarrow ins\ x\ t \in T\ h)$
proof (*induction h arbitrary: t*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc h*)
then show *?case* **by** (*fastforce simp: Bp_if_B n2_type dest: n1_type*)
qed

A detailed proof:

lemma *ins_type*:
shows $t \in B\ h \implies ins\ x\ t \in Bp\ h$ **and** $t \in U\ h \implies ins\ x\ t \in T\ h$
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
   $t1$ :  $t1 \in T\ h$  and  $t2$ :  $t2 \in T\ h$  and  $t12$ :  $t1 \in B\ h \vee t2 \in B\ h$ 
  by auto
have ?case if  $x < a$ 
proof -
  have  $n2\ (ins\ x\ t1)\ a\ t2 \in Bp\ (Suc\ h)$ 
  proof cases
    assume  $t1 \in B\ h$ 
    with  $t2$  show ?thesis by (simp add: Suc.IH(1)  $n2\_type$ )
  next
    assume  $t1 \notin B\ h$ 
    hence 1:  $t1 \in U\ h$  and 2:  $t2 \in B\ h$  using  $t1\ t12$  by auto
    show ?thesis by (metis Suc.IH(2)[OF 1]  $Bp\_if\_B$ [OF 2]  $n2\_type$ )
  qed
  with  $\langle x < a \rangle$  show ?case by simp
qed
moreover
have ?case if  $a < x$ 
proof -
  have  $n2\ t1\ a\ (ins\ x\ t2) \in Bp\ (Suc\ h)$ 
  proof cases
    assume  $t2 \in B\ h$ 
    with  $t1$  show ?thesis by (simp add: Suc.IH(1)  $n2\_type$ )
  next
    assume  $t2 \notin B\ h$ 
    hence 1:  $t1 \in B\ h$  and 2:  $t2 \in U\ h$  using  $t2\ t12$  by auto
    show ?thesis by (metis  $Bp\_if\_B$ [OF 1] Suc.IH(2)[OF 2]  $n2\_type$ )
  qed
  with  $\langle a < x \rangle$  show ?case by simp
qed
moreover
have ?case if  $x = a$ 
proof -
  from 1 have  $t \in Bp\ (Suc\ h)$  by (rule  $Bp\_if\_B$ )
  thus ?case using  $\langle x = a \rangle$  by simp
qed
ultimately show ?case by auto
next
case 2 thus ?case using Suc(1)  $n1\_type$  by fastforce }
qed

```

lemma *insert_type*:
 $t \in B\ h \implies insert\ x\ t \in B\ h \cup B\ (Suc\ h)$

unfolding *insert_def* **by** (*metis ins_type(1) tree_type*)

end

32.4.2 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 \longleftrightarrow h = 0

by (*cases h, auto*)**+**

context *delete*

begin

lemma *n2_type1*:

$\llbracket t1 \in Um\ h; t2 \in B\ h \rrbracket \implies n2\ t1\ a\ t2 \in T\ (Suc\ h)$

apply(*cases h rule: Bp.cases*)

apply *auto*[2]

apply(*erule exE bexE conjE imageE | simp | erule disjE*)**+**

done

lemma *n2_type2*:

$\llbracket t1 \in B\ h; t2 \in Um\ h \rrbracket \implies n2\ t1\ a\ t2 \in T\ (Suc\ h)$

apply(*cases h rule: Bp.cases*)

using *Um.simps(1)* **apply** *blast*

apply *force*

apply(*erule exE bexE conjE imageE | simp | erule disjE*)**+**

done

lemma *n2_type3*:

$\llbracket t1 \in T\ h; t2 \in T\ h \rrbracket \implies n2\ t1\ a\ t2 \in T\ (Suc\ h)$

apply(*cases h rule: Bp.cases*)

apply *auto*[2]

apply(*erule exE bexE conjE imageE | simp | erule disjE*)**+**

done

lemma *split_minNoneN0*: $\llbracket t \in B\ h; split_min\ t = None \rrbracket \implies t = N0$

by (*cases t*) (*auto split: option.splits*)

lemma *split_minNoneN1* : $\llbracket t \in U\ h; split_min\ t = None \rrbracket \implies t = N1\ N0$

by (*cases h*) (*auto simp: split_minNoneN0 split: option.splits*)

```

lemma split_min_type:
   $t \in B \ h \implies \text{split\_min } t = \text{Some } (a, t') \implies t' \in T \ h$ 
   $t \in U \ h \implies \text{split\_min } t = \text{Some } (a, t') \implies t' \in Um \ h$ 
proof (induction h arbitrary: t a t')
  case (Suc h)
  { case 1
    then obtain t1 a t2 where [simp]:  $t = N2 \ t1 \ a \ t2$  and
       $t12: t1 \in T \ h \ t2 \in T \ h \ t1 \in B \ h \vee t2 \in B \ h$ 
      by auto
    show ?case
    proof (cases split_min t1)
      case None
      show ?thesis
      proof cases
        assume  $t1 \in B \ h$ 
        with split_minNoneN0[OF this None] 1 show ?thesis by(auto)
      next
        assume  $t1 \notin B \ h$ 
        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  $t1 \in B \ h$ 
        from Suc.IH(1)[OF this] 1 have  $t1' \in T \ h$  by simp
        from n2_type3[OF this t12(2)] 1 show ?thesis by auto
      next
        assume  $t1 \notin B \ h$ 
        hence  $t1: t1 \in U \ h$  and  $t2: t2 \in B \ h$  using t12 by auto
        from Suc.IH(2)[OF t1] have  $t1' \in Um \ h$  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  $t1: t1 \in B \ h$  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  $t1' \in T\ h$  by simp
    thus ?thesis using 2 by auto
  qed
}
qed auto

```

lemma *del_type*:

$t \in B\ h \implies del\ x\ t \in T\ h$

$t \in U\ h \implies del\ x\ t \in Um\ h$

proof (*induction h arbitrary: x t*)

case ($Suc\ h$)

{ **case** 1

then obtain $l\ a\ r$ where $[simp]: t = N2\ l\ a\ r$ and

$lr: l \in T\ h\ r \in T\ h\ l \in B\ h \vee r \in B\ h$ by auto

have ?case if $x < a$

proof *cases*

assume $l \in B\ h$

from $n2_type3[OF\ Suc.IH(1)[OF\ this]\ lr(2)]$

show ?thesis using $\langle x < a \rangle$ by(*simp*)

next

assume $l \notin B\ h$

hence $l \in U\ h\ r \in B\ h$ using lr by auto

from $n2_type1[OF\ Suc.IH(2)[OF\ this(1)]\ this(2)]$

show ?thesis using $\langle x < a \rangle$ by(*simp*)

qed

moreover

have ?case if $x > a$

proof *cases*

assume $r \in B\ h$

from $n2_type3[OF\ lr(1)\ Suc.IH(1)[OF\ this]]$

show ?thesis using $\langle x > a \rangle$ by(*simp*)

next

assume $r \notin B\ h$

hence $l \in B\ h\ r \in U\ h$ using lr by auto

from $n2_type2[OF\ this(1)\ Suc.IH(2)[OF\ this(2)]]$

show ?thesis using $\langle x > a \rangle$ by(*simp*)

qed

moreover

have ?case if $[simp]: x = a$

proof (*cases split_min r*)

case *None*

show ?thesis

proof *cases*

```

    assume  $r \in B\ h$ 
    with split_minNoneN0[OF this None] lr show ?thesis by (simp)
next
    assume  $r \notin B\ h$ 
    hence  $r \in U\ h$  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  $r \in B\ h$ 
    from split_min_type(1)[OF this] n2_type3[OF lr(1)]
    show ?thesis by simp
next
    assume  $r \notin B\ h$ 
    hence  $l \in B\ h$  and  $r \in U\ h$  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*: $t \in T\ (h+1) \implies \text{tree } t \in B\ (h+1) \cup B\ h$
by(*auto*)

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

end

32.5 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)

```

32.6 Height-Size Relation

By Daniel Stüwe

```

fun fib_tree :: nat  $\Rightarrow$  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  $\Rightarrow$  nat where
  fib' 0 = 0
| fib' (Suc 0) = 1
| fib' (Suc(Suc h)) = 1 + fib' (Suc h) + fib' h

```

```

fun size :: 'a bro  $\Rightarrow$  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  $\in$  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  $\in$  B (Suc n)
  obtains
    (BB) t1  $\in$  B n and t2  $\in$  B n |
    (UB) t1  $\in$  U n and t2  $\in$  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

```

33 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

33.1 Functional Correctness Proofs

33.1.1 Proofs for lookup

```
lemma lookup_map_of:  $t \in T \ h \implies$   
   $sorted1(inorder\ t) \implies lookup\ t\ x = map\_of\ (inorder\ t)\ x$   
by(induction  $h$  arbitrary:  $t$ ) (auto simp: map_of_simps split: option.splits)
```

33.1.2 Proofs for update

```
context update  
begin
```

```
lemma inorder_upd:  $t \in T \ h \implies$   
   $sorted1(inorder\ t) \implies inorder(upd\ x\ y\ t) = upd\_list\ x\ y\ (inorder\ t)$   
by(induction  $h$  arbitrary:  $t$ ) (auto simp: upd_list_simps inorder_n1 inorder_n2)
```

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

```
end
```

33.1.3 Proofs for deletion

```
context delete  
begin
```

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

```
end
```

33.2 Invariant Proofs

33.2.1 Proofs for update

context *update*
begin

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

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

end

33.2.2 Proofs for deletion

context *delete*
begin

lemma *del_type*:
 $t \in B \ h \implies \text{del } x \ t \in T \ h$
 $t \in U \ h \implies \text{del } x \ t \in Um \ h$
proof (*induction h arbitrary: x t*)
 case (*Suc h*)
 { **case** 1
 then obtain *l a b r* **where** [*simp*]: $t = N2 \ l \ (a,b) \ r$ **and**
 $lr: l \in T \ h \ r \in T \ h \ l \in B \ h \vee r \in B \ h$ **by** *auto*
 have *?case* **if** $x < a$
 proof *cases*
 assume $l \in B \ h$
 from *n2_type3*[*OF Suc.IH(1)*][*OF this*] *lr*(2)
 show *?thesis* **using** $\langle x < a \rangle$ **by**(*simp*)
 next
 assume $l \notin B \ h$
 hence $l \in U \ h \ r \in B \ h$ **using** *lr* **by** *auto*
 from *n2_type1*[*OF Suc.IH(2)*][*OF this*(1)] *this*(2)
 show *?thesis* **using** $\langle x < a \rangle$ **by**(*simp*)
 qed
 moreover

```

have ?case if  $x > a$ 
proof cases
  assume  $r \in B \ h$ 
  from  $n2\_type3[OF \ lr(1) \ Suc.IH(1)[OF \ this]]$ 
  show ?thesis using  $\langle x > a \rangle$  by(simp)
next
  assume  $r \notin B \ h$ 
  hence  $l \in B \ h \ r \in U \ h$  using  $lr$  by auto
  from  $n2\_type2[OF \ this(1) \ Suc.IH(2)[OF \ this(2)]]$ 
  show ?thesis using  $\langle x > a \rangle$  by(simp)
qed
moreover
have ?case if  $[simp]: x=a$ 
proof (cases split_min  $r$ )
  case None
  show ?thesis
  proof cases
    assume  $r \in B \ h$ 
    with split_minNoneN0[OF this None]  $lr$  show ?thesis by(simp)
  next
    assume  $r \notin B \ h$ 
    hence  $r \in U \ h$  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  $r \in B \ h$ 
    from split_min_type(1)[OF this]  $n2\_type3[OF \ lr(1)]$ 
    show ?thesis by simp
  next
    assume  $r \notin B \ h$ 
    hence  $l \in B \ h$  and  $r \in U \ h$  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:
   $t \in B\ h \implies \text{delete}\ x\ t \in B\ h \cup B(h-1)$ 
unfolding delete_def
by (cases h) (simp, metis del_type(1) tree_type Suc_eq_plus1 diff_Suc_1)

end

```

33.3 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 =  $\lambda t. \exists h. t \in B\ h$ 
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

```

34 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  $\Rightarrow$  nat where
  lvl Leaf = 0 |
  lvl (Node _ (_, lv) _) = lv

```

```

fun invar :: 'a aa_tree  $\Rightarrow$  bool where
invar Leaf = True |
invar (Node l (a, h) r) =
  (invar l  $\wedge$  invar r  $\wedge$ 
   h = lvl l + 1  $\wedge$  (h = lvl r + 1  $\vee$  ( $\exists$  lr b rr. r = Node lr (b,h) rr  $\wedge$  h =
   lvl rr + 1)))

```

```

fun skew :: 'a aa_tree  $\Rightarrow$  '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  $\Rightarrow$  'a aa_tree where
split (Node t1 (a, lva) (Node t2 (b, lvb) (Node t3 (c, lvc) t4))) =
  (if lva = lvb  $\wedge$  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  $\Rightarrow$  'a aa_tree  $\Rightarrow$  '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  $\Rightarrow$  split (skew (Node (insert x t1) (a,lv) t2)) |
   GT  $\Rightarrow$  split (skew (Node t1 (a,lv) (insert x t2))) |
   EQ  $\Rightarrow$  Node t1 (x, lv) t2)

```

```

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

```

```

definition adjust :: 'a aa_tree  $\Rightarrow$  'a aa_tree where
adjust t =
  (case t of
   Node l (x,lv) r  $\Rightarrow$ 
    (if lvl l  $\geq$  lv-1  $\wedge$  lvl r  $\geq$  lv-1 then t else
     if lvl r < lv-1  $\wedge$  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 snl 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 ∧ snl
r)) ∨
  (lv = lvl l + 2 ∧ (lv = lvl r + 1 ∨ lv = lvl r ∧ snl r))))

```

```

declare pre_adjust.simps [simp del]

```

34.1 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)

```

$\text{else } t)$
 $| t \Rightarrow t)$
by(*auto split: tree.split*)

lemma skew_case: $\text{skew } t = (\text{case } t \text{ of}$
 $\text{Node (Node } t1 \text{ (y, lvy) } t2) \text{ (x, lvx) } t3 \Rightarrow$
 $(\text{if } lvx = lvy \text{ then Node } t1 \text{ (y, lvy) (Node } t2 \text{ (x, lvx) } t3) \text{ else } t)$
 $| t \Rightarrow t)$
by(*auto split: tree.split*)

lemma lvl_0_iff: $\text{invar } t \Longrightarrow \text{lvl } t = 0 \longleftrightarrow t = \text{Leaf}$
by(*cases t*) *auto*

lemma lvl_Suc_iff: $\text{lvl } t = \text{Suc } n \longleftrightarrow (\exists l a r. t = \text{Node } l \text{ (a, Suc } n) \text{ } r)$
by(*cases t*) *auto*

lemma lvl_skew: $\text{lvl } (\text{skew } t) = \text{lvl } t$
by(*cases t rule: skew.cases*) *auto*

lemma lvl_split: $\text{lvl } (\text{split } t) = \text{lvl } t \vee \text{lvl } (\text{split } t) = \text{lvl } t + 1 \wedge \text{sngl } (\text{split } t)$
by(*cases t rule: split.cases*) *auto*

lemma invar_2Nodes: $\text{invar } (\text{Node } l \text{ (x, lv) (Node } rl \text{ (rx, rlv) } rr)) =$
 $(\text{invar } l \wedge \text{invar } \langle rl, (rx, rlv), rr \rangle \wedge lv = \text{Suc } (\text{lvl } l) \wedge$
 $(lv = \text{Suc } rlv \vee rlv = lv \wedge lv = \text{Suc } (\text{lvl } rr)))$
by *simp*

lemma invar_NodeLeaf[*simp*]:
 $\text{invar } (\text{Node } l \text{ (x, lv) Leaf}) = (\text{invar } l \wedge lv = \text{Suc } (\text{lvl } l) \wedge lv = \text{Suc } 0)$
by *simp*

lemma sngl_if_invar: $\text{invar } (\text{Node } l \text{ (a, n) } r) \Longrightarrow n = \text{lvl } r \Longrightarrow \text{sngl } r$
by(*cases r rule: sngl.cases*) *clarsimp+*

34.2 Invariance

34.2.1 Proofs for insert

lemma lvl_insert_aux:
 $\text{lvl } (\text{insert } x \text{ } t) = \text{lvl } t \vee \text{lvl } (\text{insert } x \text{ } t) = \text{lvl } t + 1 \wedge \text{sngl } (\text{insert } x \text{ } t)$
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*) $lvl\ (insert\ x\ t) = lvl\ t \mid$

(*Incr*) $lvl\ (insert\ x\ t) = lvl\ t + 1\ snl\ (insert\ x\ t)$

using *lvl_insert_aux* **by** *blast*

lemma *lvl_insert_snl*: $invar\ t \implies snl\ t \implies lvl(insert\ x\ t) = lvl\ t$

proof (*induction t rule: insert.induct*)

case ($2\ x\ t1\ a\ lv\ t2$)

consider (LT) $x < a \mid (GT)\ x > a \mid (EQ)\ x = a$

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 snl.simps(3)*)**+**

qed (*auto simp add: lvl_0_iff*)

qed *simp*

qed *simp*

lemma *skew_invar*: $invar\ t \implies skew\ t = t$

by(*cases t rule: skew.cases*) *auto*

lemma *split_invar*: $invar\ t \implies split\ t = t$

by(*cases t rule: split.cases*) *clarsimp+*

lemma *invar_NodeL*:

$\llbracket invar(Node\ l\ (x,\ n)\ r); invar\ l'; lvl\ l' = lvl\ l \rrbracket \implies invar(Node\ l'\ (x,\ n)\ r)$

by(*auto*)

lemma *invar_NodeR*:

$\llbracket invar(Node\ l\ (x,\ n)\ r); n = lvl\ r + 1; invar\ r'; lvl\ r' = lvl\ r \rrbracket \implies invar(Node\ l\ (x,\ n)\ r')$

by(*auto*)

lemma *invar_NodeR2*:
 $\llbracket \text{invar}(\text{Node } l \ (x, n) \ r); \text{sngl } r'; n = \text{lvl } r + 1; \text{invar } r'; \text{lvl } r' = n \rrbracket \implies$
 $\text{invar}(\text{Node } l \ (x, n) \ r')$
by(*cases* *r'* *rule*: *sngl.cases*) *clarsimp*+

lemma *lvl_insert_incr_iff*: $(\text{lvl}(\text{insert } a \ t) = \text{lvl } t + 1) \longleftrightarrow$
 $(\exists l \ x \ r. \text{insert } a \ t = \text{Node } l \ (x, \text{lvl } t + 1) \ r \wedge \text{lvl } l = \text{lvl } r)$
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*: $\text{invar } t \implies \text{invar}(\text{insert } a \ t)$
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 $\langle a < x \rangle$ *invar_NodeL[OF N.premis iil Same]*
 by (*simp add: skew_invar split_invar del: invar.simps*)
 next
 case (*Incr*)
 then obtain *t1 w t2* **where** *ial[simp]*: $\text{insert } a \ l = \text{Node } t1 \ (w, n) \ t2$
 using *N.premis* **by** (*auto simp: lvl_Suc_iff*)
 have *l12*: $\text{lvl } t1 = \text{lvl } t2$
 by (*metis Incr(1) ial lvl_insert_incr_iff tree.inject*)
 have $\text{insert } a \ ?t = \text{split}(\text{skew}(\text{Node } (\text{insert } a \ l) \ (x, n) \ r))$
 by(*simp add: \langle a < x \rangle*)
 also have $\text{skew}(\text{Node } (\text{insert } a \ l) \ (x, n) \ r) = \text{Node } t1 \ (w, n) \ (\text{Node } t2 \ (x, n) \ r)$
 by(*simp*)
 also have *invar*(*split ...*)
 proof (*cases* *r* *rule*: *tree2_cases*)
 case *Leaf*
 hence $l = \text{Leaf}$ **using** *N.premis* **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.prem1 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.prem1 0 by (auto simp: lvl_Suc_iff)
      from N.prem1 iar 0 iir
      show ?thesis by (auto simp: split_case split: tree.splits)
    qed
  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.prem1 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
  qed
  moreover
  have  $a = x \implies ?case$  using  $N.prem$ s by auto
  ultimately show  $?case$  by blast
qed simp

```

34.2.2 Proofs for delete

```

lemma invarL: ASSUMPTION(invar  $\langle l, (a, lv), r \rangle \implies \textit{invar } l$ 
by(simp add: ASSUMPTION_def)

```

```

lemma invarR: ASSUMPTION(invar  $\langle l, (a, lv), r \rangle \implies \textit{invar } r$ 
by(simp add: ASSUMPTION_def)

```

```

lemma sngl_NodeI:
  sngl (Node l (a,lv) r)  $\implies$  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  $\wedge$  invar r  $\wedge$ 
     $lv = \textit{Suc } (lv \textit{ l } r) \wedge lv \textit{ l} = lv \textit{ r} \mid$ 
  (tDouble) invar l  $\wedge$  invar r  $\wedge$ 
     $lv = lv \textit{ r} \wedge \textit{Suc } (lv \textit{ l}) = lv \textit{ r} \wedge \textit{sngl } r \mid$ 
  (rDown) invar l  $\wedge$  invar r  $\wedge$ 
     $lv = \textit{Suc } (\textit{Suc } (lv \textit{ l } r)) \wedge lv = \textit{Suc } (lv \textit{ l}) \mid$ 
  (lDown_tSngl) invar l  $\wedge$  invar r  $\wedge$ 
     $lv = \textit{Suc } (lv \textit{ r}) \wedge lv = \textit{Suc } (\textit{Suc } (lv \textit{ l})) \mid$ 
  (lDown_tDouble) invar l  $\wedge$  invar r  $\wedge$ 
     $lv = lv \textit{ r} \wedge lv = \textit{Suc } (\textit{Suc } (lv \textit{ l})) \wedge \textit{sngl } r$ 
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))  $\vee$  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 $t' \wedge$
 $(lvl\ t' = lvl\ t \vee lvl\ t' + 1 = lvl\ t) \wedge$
 $(lvl\ t' = lvl\ t \wedge snl\ t \longrightarrow snl\ t')$

lemma *pre_adj_if_postR*:

invar $\langle lv, (l, a), r \rangle \implies post_del\ r\ r' \implies pre_adjust\ \langle lv, (l, a), r \rangle$

by (*cases* *snl* r)

(*auto simp*: *pre_adjust.simps* *post_del_def* *invar.simps*(2) *elim*: *snl.elims*)

lemma *pre_adj_if_postL*:

invar $\langle l, (a, lv), r \rangle \implies post_del\ l\ l' \implies pre_adjust\ \langle l', (b, lv), r \rangle$

by (*cases* *snl* r)

(*auto simp*: *pre_adjust.simps* *post_del_def* *invar.simps*(2) *elim*: *snl.elims*)

lemma *post_del_adjL*:

$\llbracket invar\ \langle l, (a, lv), r \rangle; pre_adjust\ \langle l', (b, lv), r \rangle \rrbracket$
 $\implies post_del\ \langle l, (a, lv), r \rangle\ (adjust\ \langle l', (b, lv), r \rangle)$

unfolding *post_del_def*

by (*metis* *invar_adjust* *lvl_adjust* *snl_NodeI* *snl_adjust* *lvl.simps*(2))

lemma *post_del_adjR*:

assumes *invar* $\langle l, (a, lv), r \rangle$ *pre_adjust* $\langle l, (a, lv), r \rangle$ *post_del* $r\ r'$

shows *post_del* $\langle l, (a, lv), r \rangle\ (adjust\ \langle l, (a, lv), r \rangle)$

proof (*unfold* *post_del_def*, *safe del*: *disjCI*)

let $?t = \langle l, (a, lv), r \rangle$

let $?t' = adjust\ \langle l, (a, lv), r \rangle$

show *invar* $?t'$ **by** (*rule* *invar_adjust* [*OF* *assms*(2)])

show $lvl\ ?t' = lvl\ ?t \vee lvl\ ?t' + 1 = lvl\ ?t$

using *lvl_adjust* [*OF* *assms*(2)] **by** *auto*

show *snl* $?t'$ **if** *as*: $lvl\ ?t' = lvl\ ?t\ snl\ ?t$

proof –

have *s*: *snl* $\langle l, (a, lv), r \rangle$

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) *snl_adjust* [*OF* *assms*(2) *s*] **by** *simp*

qed

qed

declare *prod.splits* [*split*]

```

theorem post_split_max:
   $\llbracket \text{invar } t; (t', x) = \text{split\_max } t; t \neq \text{Leaf} \rrbracket \implies \text{post\_del } t \ t'$ 
proof (induction t arbitrary: t' rule: split_max.induct)
  case ( $2 \ l \ a \ lv \ rl \ bl \ rr$ )
  let  $?r = \langle rl, bl, rr \rangle$ 
  let  $?t = \langle l, (a, lv), ?r \rangle$ 
  from  $2.\text{prems}(2)$  obtain  $r'$  where  $r': (r', x) = \text{split\_max } ?r$ 
  and  $[\text{simp}]: t' = \text{adjust } \langle l, (a, lv), r' \rangle$  by auto
  from  $2.IH[OF \_ r'] \langle \text{invar } ?t \rangle$  have  $\text{post}: \text{post\_del } ?r \ r'$  by simp
  note  $\text{preR} = \text{pre\_adj\_if\_postR}[OF \langle \text{invar } ?t \rangle \text{ post}]$ 
  show  $?case$  by (simp add: post\_del\_adjR $[OF \ 2.\text{prems}(1) \ \text{preR} \ \text{post}]$ )
qed (auto simp: post\_del\_def)

theorem post_delete:  $\text{invar } t \implies \text{post\_del } t \ (\text{delete } x \ t)$ 
proof (induction t rule: tree2_induct)
  case (Node l a lv r)

  let  $?l' = \text{delete } x \ l$  and  $?r' = \text{delete } x \ r$ 
  let  $?t = \text{Node } l \ (a, lv) \ r$  let  $?t' = \text{delete } x \ ?t$ 

  from Node.prems have  $\text{inv\_l}: \text{invar } l$  and  $\text{inv\_r}: \text{invar } r$  by (auto)

  note  $\text{post\_l}' = \text{Node.IH}(1)[OF \ \text{inv\_l}]$ 
  note  $\text{preL} = \text{pre\_adj\_if\_postL}[OF \ \text{Node.prems} \ \text{post\_l}']$ 

  note  $\text{post\_r}' = \text{Node.IH}(2)[OF \ \text{inv\_r}]$ 
  note  $\text{preR} = \text{pre\_adj\_if\_postR}[OF \ \text{Node.prems} \ \text{post\_r}']$ 

  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
 $\text{post\_r}'$ )
  next
    case equal
    show  $?thesis$ 
    proof cases
      assume  $l = \text{Leaf}$  thus  $?thesis$  using equal Node.prems
      by (auto simp: post\_del\_def invar.simps(2))
    next

```

```

      assume  $l \neq \text{Leaf}$  thus ?thesis using equal
      by simp (metis Node.premis 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]

```

34.3 Functional Correctness

34.3.1 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)  $\implies$  inorder(insert x t) = ins_list x (inorder t)
by(induction t) (auto simp: ins_list_simps inorder_split inorder_skew)

```

34.3.2 Proofs for delete

```

lemma inorder_adjust:  $t \neq \text{Leaf} \implies \text{pre\_adjust } t \implies \text{inorder}(\text{adjust } t)
= \text{inorder } t$ 
by(cases t)
  (auto simp: adjust_def inorder_skew inorder_split invar_simps(2) pre_adjust_simps
split: tree.splits)

```

```

lemma split_maxD:
   $\llbracket \text{split\_max } t = (t', x); t \neq \text{Leaf}; \text{invar } t \rrbracket \implies \text{inorder } t' @ [x] = \text{inorder } t$ 
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:
   $\text{invar } t \implies \text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{delete } x t) = \text{del\_list } x (\text{inorder } t)$ 
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

```

35 AA Tree Implementation of Maps

```

theory AA_Map

```

```

imports

```

```

  AA_Set

```

```

  Lookup2

```

```

begin

```

```

fun update :: 'a::linorder  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) aa_tree  $\Rightarrow$  ('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  $\Rightarrow$  split (skew (Node (update x y t1) ((a,b), lv) t2)) |

```

```

      GT  $\Rightarrow$  split (skew (Node t1 ((a,b), lv) (update x y t2))) |

```

```

      EQ  $\Rightarrow$  Node t1 ((x,y), lv) t2)

```

```

fun delete :: 'a::linorder  $\Rightarrow$  ('a*'b) aa_tree  $\Rightarrow$  ('a*'b) aa_tree where

```

```

  delete _ Leaf = Leaf |

```

```

  delete x (Node l ((a,b), lv) r) =

```

```

    (case cmp x a of

```


$$\begin{aligned}
LT &\Rightarrow \text{adjust } (\text{Node } (\text{delete } x \ l) \ ((a,b), \ lv) \ r) \mid \\
GT &\Rightarrow \text{adjust } (\text{Node } l \ ((a,b), \ lv) \ (\text{delete } x \ r)) \mid \\
EQ &\Rightarrow (\text{if } l = \text{Leaf} \text{ then } r \\
&\quad \text{else let } (l', ab') = \text{split_max } l \text{ in adjust } (\text{Node } l' \ (ab', \ lv) \ r)))
\end{aligned}$$

35.1 Invariance

35.1.1 Proofs for insert

lemma *lvl_update_aux*:

lvl (update x y t) = lvl t \vee lvl (update x y t) = lvl t + 1 \wedge 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 \mid

(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 \implies sngl t \implies 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 \mid (GT) x > a \mid (EQ) x = a*

using *less_linear* **by** *blast*

thus *?thesis* **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)  $\longleftrightarrow$ 
  ( $\exists l\ x\ r.$  update a b t = Node l (x, lvl t + 1) r  $\wedge$  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  $\implies$  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  $\vee$  a = x  $\vee$  x < a by auto
  moreover
  have ?case if a < x
  proof (cases rule: lvl_update[of a b l])
    case (Same) thus ?thesis
      using  $\langle a < x \rangle$  invar_NodeL[OF N.prem1 iil Same]
      by (simp add: skew_invar split_invar del: invar.sims)
  next
  case (Incr)
  then obtain t1 w t2 where ial[simp]: update a b l = Node t1 (w, n) t2
    using N.prem2 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:  $\langle a < x \rangle$ )
  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.prem3 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  $m \neq n$  thus ?thesis using  $N(3)$  il l12 by(auto)
    qed
  qed
  finally show ?thesis .
qed
moreover
have ?case if  $x < a$ 
proof -
  from  $\langle \text{invar } ?t \rangle$  have  $n = \text{lvl } r \vee n = \text{lvl } r + 1$  by auto
  thus ?case
  proof
    assume 0:  $n = \text{lvl } r$ 
    have  $\text{update } a \ b \ ?t = \text{split}(\text{skew}(\text{Node } l \ (xy, n) \ (\text{update } a \ b \ r)))$ 
      using  $\langle a > x \rangle$  by(auto)
    also have  $\text{skew}(\text{Node } l \ (xy, n) \ (\text{update } a \ b \ r)) = \text{Node } l \ (xy, n) \ (\text{update } a \ b \ r)$ 
      using  $N.\text{prems}$  by(simp add: skew_case split: tree.split)
    also have  $\text{invar}(\text{split } \dots)$ 
  proof -
    from  $\text{lvl\_update\_sngl}[OF \ \text{ir\_sngl\_if\_invar}[OF \ \langle \text{invar } ?t \rangle \ 0], \text{ of } a \ b]$ 
    obtain  $t1 \ p \ t2$  where  $\text{iar}: \text{update } a \ b \ r = \text{Node } t1 \ (p, n) \ t2$ 
      using  $N.\text{prems} \ 0$  by (auto simp: lvl_Suc_iff)
    from  $N.\text{prems} \ \text{iar} \ 0 \ \text{iir}$ 
    show ?thesis by (auto simp: split_case split: tree.splits)
  qed
  finally show ?thesis .
next
  assume 1:  $n = \text{lvl } r + 1$ 
  hence  $\text{sngl } ?t$  by(cases r) auto
  show ?thesis
  proof (cases rule: lvl_update[of a b r])
    case (Same)
    show ?thesis using  $\langle x < a \rangle$  il ir invar_NodeR[OF  $N.\text{prems} \ 1 \ \text{iir} \ \text{Same}$ ]
      by (auto simp add: skew_invar split_invar)
  next
    case (Incr)
    thus ?thesis using invar_NodeR2[OF  $\langle \text{invar } ?t \rangle \ \text{Incr}(2) \ 1 \ \text{iir}$ ] 1  $\langle x < a \rangle$ 
      by (auto simp add: skew_invar split_invar split: if_splits)
  qed
qed
qed
moreover

```

```

  have  $a = x \implies ?case$  using  $N.prem s$  by  $auto$ 
  ultimately show  $?case$  by  $blast$ 
qed  $simp$ 

```

35.1.2 Proofs for delete

```

declare  $invar.simps(2)[simp\ del]$ 

```

```

theorem  $post\_delete: invar\ t \implies post\_del\ t\ (delete\ x\ t)$ 

```

```

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  $Node.prem s$  have  $inv\_l: invar\ l$  and  $inv\_r: invar\ r$  by ( $auto$ )

```

```

  note  $post\_l' = Node.IH(1)[OF\ inv\_l]$ 
  note  $preL = pre\_adj\_if\_postL[OF\ Node.prem s\ post\_l']$ 

```

```

  note  $post\_r' = Node.IH(2)[OF\ inv\_r]$ 
  note  $preR = pre\_adj\_if\_postR[OF\ Node.prem s\ post\_r']$ 

```

```

  show  $?case$ 

```

```

  proof ( $cases\ rule: linorder\_cases[of\ x\ a]$ )

```

```

    case  $less$ 

```

```

    thus  $?thesis$  using  $Node.prem s$  by ( $simp\ add: post\_del\_adjL\ preL$ )

```

```

  next

```

```

    case  $greater$ 

```

```

    thus  $?thesis$  using  $Node.prem s\ preR$  by ( $simp\ add: post\_del\_adjR\ post\_r'$ )

```

```

  next

```

```

    case  $equal$ 

```

```

    show  $?thesis$ 

```

```

    proof  $cases$ 

```

```

      assume  $l = Leaf$  thus  $?thesis$  using  $equal\ Node.prem s$ 

```

```

      by ( $auto\ simp: post\_del\_def\ invar.simps(2)$ )

```

```

    next

```

```

      assume  $l \neq Leaf$  thus  $?thesis$  using  $equal\ Node.prem s$ 

```

```

      by  $simp\ (metis\ inv\_l\ post\_del\_adjL\ post\_split\_max\ pre\_adj\_if\_postL)$ 

```

```

    qed

```

```

  qed

```

qed (*simp add: post_del_def*)

35.2 Functional Correctness Proofs

theorem *inorder_update*:

sorted1(inorder t) \implies 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*:

$\llbracket \text{invar } t; \text{sorted1}(\text{inorder } t) \rrbracket \implies$
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

36 Join-Based Implementation of Sets

theory *Set2_Join*

imports

Isin2

begin

This theory implements the set operations *insert*, *delete*, *union*, *intersection* and *difference*. 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  $\Rightarrow$  'a  $\Rightarrow$  ('a*'b) tree  $\Rightarrow$  ('a*'b) tree
fixes inv :: ('a*'b) tree  $\Rightarrow$  bool
assumes set_join: set_tree (join l a r) = set_tree l  $\cup$  {a}  $\cup$  set_tree r
assumes bst_join: bst (Node l (a, b) r)  $\implies$  bst (join l a r)
assumes inv_Leaf: inv  $\langle$ 
assumes inv_join:  $\llbracket$  inv l; inv r  $\rrbracket \implies$  inv (join l a r)
assumes inv_Node:  $\llbracket$  inv (Node l (a,b) r)  $\rrbracket \implies$  inv l  $\wedge$  inv r
begin

```

```

declare set_join [simp] Let_def[simp]

```

36.1 split_min

```

fun split_min :: ('a*'b) tree  $\Rightarrow$  'a  $\times$  ('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:
 $\llbracket$  split_min t = (m,t'); t  $\neq$  Leaf  $\rrbracket \implies$  m  $\in$  set_tree t  $\wedge$  set_tree t =
{m}  $\cup$  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:
 $\llbracket$  split_min t = (m,t'); bst t; t  $\neq$  Leaf  $\rrbracket \implies$  bst t'  $\wedge$  ( $\forall x \in$  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*:

$\llbracket \text{split_min } t = (m, t'); \text{ inv } t; t \neq \text{Leaf} \rrbracket \implies \text{inv } t'$

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

36.2 *join2*

definition *join2* :: ('a*'b) *tree* \Rightarrow ('a*'b) *tree* \Rightarrow ('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* \cup *set_tree* *r*

by(*cases* *r*)(*simp_all add*: *split_min_set join2_def split*: *prod.split*)

lemma *bst_join2*: $\llbracket \text{bst } l; \text{bst } r; \forall x \in \text{set_tree } l. \forall y \in \text{set_tree } r. x < y \rrbracket$
 $\implies \text{bst } (\text{join2 } l \ r)$

by(*cases* *r*)(*simp_all add*: *bst_join split_min_set split_min_bst join2_def*
split: *prod.split*)

lemma *inv_join2*: $\llbracket \text{inv } l; \text{inv } r \rrbracket \implies \text{inv } (\text{join2 } l \ r)$

by(*cases* *r*)(*simp_all add*: *inv_join split_min_set split_min_inv join2_def*
split: *prod.split*)

36.3 *split*

fun *split* :: 'a \Rightarrow ('a*'b)*tree* \Rightarrow ('a*'b)*tree* \times *bool* \times ('a*'b)*tree* **where**

split *x* *Leaf* = (*Leaf*, *False*, *Leaf*) |

split *x* (*Node* *l* (*a*, $_$) *r*) =

(*case cmp* *x* *a* of

LT \Rightarrow let (*l1*, *b*, *l2*) = *split* *x* *l* in (*l1*, *b*, *join* *l2* *a* *r*) |

GT \Rightarrow let (*r1*, *b*, *r2*) = *split* *x* *r* in (*join* *l* *a* *r1*, *b*, *r2*) |

EQ \Rightarrow (*l*, *True*, *r*))

lemma *split*: *split* *x* *t* = (*l*, *b*, *r*) $\implies \text{bst } t \implies$

set_tree *l* = {*a* \in *set_tree* *t*. *a* < *x*} \wedge *set_tree* *r* = {*a* \in *set_tree* *t*. *x* < *a*}

\wedge (*b* = (*x* \in *set_tree* *t*)) $\wedge \text{bst } l \wedge \text{bst } r$

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 x y
    and split x ⟨y, (a, b), z⟩ = (l1, xin, join l2 a z) and cmp x a = LT
  | (GT) r1 xin r2 where (r1,xin,r2) = split x z
    and split x ⟨y, (a, b), z⟩ = (join y a r1, xin, r2) and cmp x a = GT
  | (EQ) split x ⟨y, (a, b), z⟩ = (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 x y⟩[symmetric]] Node.prem
  show ?thesis by (force intro!: bst_join)
next
  case (GT r1 xin r2)
  with Node.IH(2)[OF ⟨(r1,xin,r2) = split x z⟩[symmetric]] Node.prem
  show ?thesis by (force intro!: bst_join)
next
  case EQ
  with Node.prem show ?thesis by auto
qed
qed

lemma split_inv: split x t = (l,b,r)  $\implies$  inv t  $\implies$  inv l  $\wedge$  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]

```

36.4 insert

definition insert :: 'a \Rightarrow ('a*'b) tree \Rightarrow ('a*'b) tree **where**
 insert x t = (let (l,_,r) = split x t in join l x r)

lemma set_tree_insert: bst t \implies set_tree (insert x t) = {x} \cup set_tree t
by(auto simp add: insert_def split split: prod.split)

lemma bst_insert: bst t \implies bst (insert x t)

by(*auto simp add: insert_def bst_join dest: split split: prod.split*)

lemma *inv_insert*: $\text{inv } t \implies \text{inv } (\text{insert } x \ t)$

by(*force simp: insert_def inv_join dest: split_inv split: prod.split*)

36.5 delete

definition *delete* :: $'a \Rightarrow ('a * 'b) \text{ tree} \Rightarrow ('a * 'b) \text{ tree}$ **where**

delete $x \ t = (\text{let } (l, _, r) = \text{split } x \ t \text{ in } \text{join2 } l \ r)$

lemma *set_tree_delete*: $\text{bst } t \implies \text{set_tree } (\text{delete } x \ t) = \text{set_tree } t - \{x\}$

by(*auto simp: delete_def split split: prod.split*)

lemma *bst_delete*: $\text{bst } t \implies \text{bst } (\text{delete } x \ t)$

by(*force simp add: delete_def intro: bst_join2 dest: split split: prod.split*)

lemma *inv_delete*: $\text{inv } t \implies \text{inv } (\text{delete } x \ t)$

by(*force simp: delete_def inv_join2 dest: split_inv split: prod.split*)

36.6 union

fun *union* :: $('a * 'b) \text{ tree} \Rightarrow ('a * 'b) \text{ tree} \Rightarrow ('a * 'b) \text{ tree}$ **where**

union $t1 \ t2 =$

(*if* $t1 = \text{Leaf}$ *then* $t2$ *else*

if $t2 = \text{Leaf}$ *then* $t1$ *else*

case $t1$ *of* $\text{Node } l1 \ (a, _) \ r1 \Rightarrow$

let $(l2, _, r2) = \text{split } a \ t2;$

$l' = \text{union } l1 \ l2; \ r' = \text{union } r1 \ r2$

in $\text{join } l' \ a \ r')$

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

lemma *set_tree_union*: $\text{bst } t2 \implies \text{set_tree } (\text{union } t1 \ t2) = \text{set_tree } t1 \cup \text{set_tree } t2$

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*: $\llbracket \text{bst } t1; \text{bst } t2 \rrbracket \implies \text{bst } (\text{union } t1 \ t2)$

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:  $\llbracket \text{inv } t1; \text{inv } t2 \rrbracket \implies \text{inv } (\text{union } t1 \ t2)$ 
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

```

36.7 inter

```

fun inter :: ('a*'b)tree  $\Rightarrow$  ('a*'b)tree  $\Rightarrow$  ('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  $\Rightarrow$ 
    let (l2,b,r2) = split a t2;
    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:
 $\llbracket \text{bst } t1; \text{bst } t2 \rrbracket \implies \text{set\_tree } (\text{inter } t1 \ t2) = \text{set\_tree } t1 \cap \text{set\_tree } t2$ 
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 *:  $a \notin ?L1 \cup ?R1$  using  $\langle \text{bst } t1 \rangle$  by (fastforce)
      obtain l2 b r2 where sp: split a t2 = (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.premis(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.premis(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

```

36.8 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 a t1;
       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 a t1 = (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 ∉ ?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.prem1(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.prem1(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:  $\llbracket \text{bst } t1; \text{bst } t2 \rrbracket \implies \text{bst } (\text{diff } t1 \ t2)$ 
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:  $\llbracket \text{inv } t1; \text{inv } t2 \rrbracket \implies \text{inv } (\text{diff } t1 \ t2)$ 
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 =  $\lambda t. \text{inv } t \wedge \text{bst } t$ 
proof (standard, goal_cases)
  case 1 show ?case by (simp)
next
  case 2 thus ?case by(simp add: isin_set_tree)
next
  case 3 thus ?case by (simp add: set_tree_insert)
next
  case 4 thus ?case by (simp add: set_tree_delete)
next
  case 5 thus ?case by (simp add: inv_Leaf)
next
  case 6 thus ?case by (simp add: bst_insert inv_insert)
next
  case 7 thus ?case by (simp add: bst_delete inv_delete)
next
  case 8 thus ?case by(simp add: set_tree_union)
next

```

```

    case 9 thus ?case by (simp add: set_tree_inter)
next
    case 10 thus ?case by (simp add: set_tree_diff)
next
    case 11 thus ?case by (simp add: bst_union inv_union)
next
    case 12 thus ?case by (simp add: bst_inter inv_inter)
next
    case 13 thus ?case by (simp add: bst_diff inv_diff)
qed

```

end

```

interpretation unbal: Set2_Join
where join =  $\lambda l\ x\ r.$  Node  $l\ (x, ())\ r$  and inv =  $\lambda t.$  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

37 Join-Based Implementation of Sets via RBTs

```

theory Set2_Join_RBT
imports
  Set2_Join
  RBT_Set
begin

```

37.1 Code

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

```

fun joinL :: 'a rbt  $\Rightarrow$  'a  $\Rightarrow$  'a rbt  $\Rightarrow$  'a rbt where

```

```

joinL l x r =
  (if bheight l ≥ bheight r then R l x r
   else case r of
     B l' x' r' ⇒ baliL (joinL l x l') x' r' |
     R l' x' r' ⇒ R (joinL l x l') x' r')

fun joinR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where
joinR l x r =
  (if bheight l ≤ bheight r then R l x r
   else case l of
     B l' x' r' ⇒ baliR l' x' (joinR r' x r) |
     R l' x' r' ⇒ R l' x' (joinR r' x r))

```

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]

```

37.2 Properties

37.2.1 Color and height invariants

lemma invc2_joinL:

```

[[ invc l; invc r; bheight l ≤ bheight r ]] ⇒
  invc2 (joinL l x r)
  ∧ (bheight l ≠ bheight r ∧ color r = Black ⟶ invc(joinL l x r))
proof (induct l x r rule: joinL.induct)
  case (1 l x r) thus ?case
  by(auto simp: invc_baliL invc2I joinL.simps[of l x r] split!: tree.splits
if_splits)
qed

```

lemma invc2_joinR:

```

[[ invc l; invh l; invc r; invh r; bheight l ≥ bheight r ]] ⇒
  invc2 (joinR l x r)
  ∧ (bheight l ≠ bheight r ∧ color l = Black ⟶ invc(joinR l x r))
proof (induct l x r rule: joinR.induct)
  case (1 l x r) thus ?case

```

by(*fastforce simp: invc_baliR invc2I joinR.simps[of l x r] split!: tree.splits if_splits*)
qed

lemma *bheight_joinL*:
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l \leq \text{bheight } r \rrbracket \implies \text{bheight } (\text{joinL } l \ x \ r) = \text{bheight } r$
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*:
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l \leq \text{bheight } r \rrbracket \implies \text{invh } (\text{joinL } l \ x \ r)$
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*:
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l \geq \text{bheight } r \rrbracket \implies \text{bheight } (\text{joinR } l \ x \ r) = \text{bheight } l$
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*:
 $\llbracket \text{invh } l; \text{invh } r; \text{bheight } l \geq \text{bheight } r \rrbracket \implies \text{invh } (\text{joinR } l \ x \ r)$
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*: $\llbracket \text{invc } l; \text{invc } r; \text{invh } l; \text{invh } r; \text{bheight } l \leq \text{bheight } r \rrbracket \implies \text{invc2 } (\text{joinL } l \ x \ r) \wedge (\text{bheight } l \neq \text{bheight } r \wedge \text{color } r = \text{Black} \longrightarrow \text{invc } (\text{joinL } l \ x \ r))$
 $\wedge \text{invh } (\text{joinL } l \ x \ r) \wedge \text{bheight } (\text{joinL } l \ x \ r) = \text{bheight } r$
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*: $\llbracket \text{inv } l; \text{inv } r; \text{invh } l; \text{invh } r; \text{bheight } l \geq \text{bheight } r \rrbracket$
 $\implies \text{inv2 } (\text{joinR } l \ x \ r) \wedge (\text{bheight } l \neq \text{bheight } r \wedge \text{color } l = \text{Black} \longrightarrow$
 $\text{inv } (\text{joinR } l \ x \ r))$
 $\wedge \text{invh } (\text{joinR } l \ x \ r) \wedge \text{bheight } (\text{joinR } l \ x \ r) = \text{bheight } l$
proof (*induct l x r rule: joinR.induct*)
case (*1 l x r*) **thus** ?*case*
by(*auto simp: inv_baliR inv2I joinR.simps[of l x r] split!: tree.splits*
if_splits)
qed

lemma *rbt_join*: $\llbracket \text{inv } l; \text{invh } l; \text{inv } r; \text{invh } r \rrbracket \implies \text{rbt}(\text{join } l \ x \ r)$
by(*simp add: inv_joinL inv_joinR invh_paint rb_def color_paint_Black*
join_def)

To make sure the the black height is not increased unnecessarily:

lemma *bheight_paint_Black*: $\text{bheight}(\text{paint } \text{Black } t) \leq \text{bheight } t + 1$
by(*cases t*) *auto*

lemma $\llbracket \text{rbt } l; \text{rbt } r \rrbracket \implies \text{bheight}(\text{join } l \ x \ r) \leq \max (\text{bheight } l) (\text{bheight } r)$
 $+ 1$
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 rb_def join_def*)

37.2.2 Inorder properties

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

lemma *inorder_joinL*: $\text{bheight } l \leq \text{bheight } r \implies \text{inorder}(\text{joinL } l \ x \ r) =$
 $\text{inorder } l @ x \# \text{inorder } r$
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*:
 $\text{inorder}(\text{joinR } l \ x \ r) = \text{inorder } l @ x \# \text{inorder } r$
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])

```

37.2.3 Set and bst properties

```

lemma set_baliL:
  set_tree(baliL l a r) = set_tree l ∪ {a} ∪ set_tree r
by(cases (l,a,r) rule: baliL.cases) (auto)

```

```

lemma set_joinL:
  bheight l ≤ bheight r ⟹ set_tree (joinL l x r) = set_tree l ∪ {x} ∪
set_tree r
proof(induction l x r rule: joinL.induct)
  case (1 l x r)
  thus ?case by(auto simp: set_baliL joinL.simps[of l x r] split!: tree.splits
color.splits)
qed

```

```

lemma set_baliR:
  set_tree(baliR l a r) = set_tree l ∪ {a} ∪ set_tree r
by(cases (l,a,r) rule: baliR.cases) (auto)

```

```

lemma set_joinR:
  set_tree (joinR l x r) = set_tree l ∪ {x} ∪ set_tree r
proof(induction l x r rule: joinR.induct)
  case (1 l x r)
  thus ?case by(force simp: set_baliR joinR.simps[of l x r] split!: tree.splits
color.splits)
qed

```

```

lemma set_paint: set_tree (paint c t) = set_tree t
by (cases t) auto

```

```

lemma set_join: set_tree (join l x r) = set_tree l ∪ {x} ∪ set_tree r
by(simp add: set_joinL set_joinR set_paint join_def)

```

```

lemma bst_baliL:

```

$\llbracket \text{bst } l; \text{bst } r; \forall x \in \text{set_tree } l. x < a; \forall x \in \text{set_tree } r. a < x \rrbracket$
 $\implies \text{bst } (\text{baliL } l \ a \ r)$
by(cases (l,a,r) rule: baliL.cases) (auto simp: ball_Un)

lemma bst_baliR:
 $\llbracket \text{bst } l; \text{bst } r; \forall x \in \text{set_tree } l. x < a; \forall x \in \text{set_tree } r. a < x \rrbracket$
 $\implies \text{bst } (\text{baliR } l \ a \ r)$
by(cases (l,a,r) rule: baliR.cases) (auto simp: ball_Un)

lemma bst_joinL:
 $\llbracket \text{bst } (\text{Node } l \ (a, n) \ r); \text{bheight } l \leq \text{bheight } r \rrbracket$
 $\implies \text{bst } (\text{joinL } l \ a \ r)$
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:
 $\llbracket \text{bst } l; \text{bst } r; \forall x \in \text{set_tree } l. x < a; \forall y \in \text{set_tree } r. a < y \rrbracket$
 $\implies \text{bst } (\text{joinR } l \ a \ r)$
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:
 $\text{bst } (\text{Node } l \ (a, n) \ r) \implies \text{bst } (\text{join } l \ a \ r)$
by(auto simp: bst_paint bst_joinL bst_joinR join_def)

lemma inv_join: $\llbracket \text{invc } l; \text{invh } l; \text{invc } r; \text{invh } r \rrbracket \implies \text{invc}(\text{join } l \ x \ r) \wedge$
 $\text{invh}(\text{join } l \ x \ r)$
by (simp add: inv_joinL inv_joinR invh_paint join_def)

37.2.4 Interpretation of *Set2_Join* with Red-Black Tree

```

global_interpretation RBT: Set2_Join
where join = join and inv =  $\lambda t. \text{invc } t \wedge \text{invh } t$ 
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
and inter_rbt = RBT.inter and union_rbt = RBT.union and diff_rbt =
RBT.diff
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  $\Rightarrow$  nat  $\Rightarrow$  'a
  fixes update :: nat  $\Rightarrow$  'a  $\Rightarrow$  'ar  $\Rightarrow$  'ar
  fixes len :: 'ar  $\Rightarrow$  nat
  fixes array :: 'a list  $\Rightarrow$  'ar

  fixes list :: 'ar  $\Rightarrow$  'a list
  fixes invar :: 'ar  $\Rightarrow$  bool

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

  assumes invar_update: invar ar  $\Longrightarrow$  n < len ar  $\Longrightarrow$  invar(update n x ar)

```

```

assumes invar_array: invar(array xs)

locale Array_Flex = Array +
fixes add_lo :: 'a  $\Rightarrow$  'ar  $\Rightarrow$  'ar
fixes del_lo :: 'ar  $\Rightarrow$  'ar
fixes add_hi :: 'a  $\Rightarrow$  'ar  $\Rightarrow$  'ar
fixes del_hi :: 'ar  $\Rightarrow$  'ar

assumes add_lo: invar ar  $\implies$  list(add_lo a ar) = a # list ar
assumes del_lo: invar ar  $\implies$  list(del_lo ar) = tl (list ar)
assumes add_hi: invar ar  $\implies$  list(add_hi a ar) = list ar @ [a]
assumes del_hi: invar ar  $\implies$  list(del_hi ar) = butlast (list ar)

assumes invar_add_lo: invar ar  $\implies$  invar (add_lo a ar)
assumes invar_del_lo: invar ar  $\implies$  invar (del_lo ar)
assumes invar_add_hi: invar ar  $\implies$  invar (add_hi a ar)
assumes invar_del_hi: invar ar  $\implies$  invar (del_hi ar)

end

```

38 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  $\Rightarrow$  bool where
  braun Leaf = True |
  braun (Node l x r) = ((size l = size r  $\vee$  size l = size r + 1)  $\wedge$  braun l  $\wedge$ 
braun r)

```

```

lemma braun_Node':
  braun (Node l x r) = (size r  $\leq$  size l  $\wedge$  size l  $\leq$  size r + 1  $\wedge$  braun l  $\wedge$ 
braun r)
by auto

```

The shape of a Braun-tree is uniquely determined by its size:

```

lemma braun_unique:  $\llbracket$  braun (t1::unit tree); braun t2; size t1 = size t2  $\rrbracket$ 
 $\implies$  t1 = t2
proof (induction t1 arbitrary: t2)
  case Leaf thus ?case by simp
next

```

```

case (Node l1 _ r1)
from Node.premis(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.premis have size l1 = size l2 ∧ size r1 = size r2 by auto
thus ?case using Node.premis(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

```

38.1 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  $x \text{ div } 2 \in \{a..b\}$  using * by auto
  moreover have  $x \in \{2 * (x \text{ div } 2), \text{Suc}(2 * (x \text{ div } 2))\}$  by auto
  ultimately show ?thesis by blast
qed
thus ?r  $\subseteq$  ?l by fastforce
qed
case (Node l x r)
hence  $\text{size } l = \text{size } r \vee \text{size } l = \text{size } r + 1$  (is ?A  $\vee$  ?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:  $(*) \ 2 \nmid A \cap \text{Suc} \nmid B = \{\}$ 
  using double_not_eq_Suc_double by auto

```

```

lemma card_braun_indices:  $\text{card } (\text{braun\_indices } t) = \text{size } t$ 
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:  $\text{braun\_indices } t = \{m..n\}$ 
  shows  $\{m..n\} = \{1..\text{size } t\}$ 
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:  $m \leq 1$ 
  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  $S :: \text{nat set}$ 
  assumes  $S = \{m..n\} \cap \{i. \text{even } i\}$ 
  shows  $\exists m' n'. S = (\lambda i. i * 2) \text{ `` } \{m'..n'\}$ 
proof –
  have  $S = (\lambda i. i * 2) \text{ `` } \{\text{Suc } m \text{ div } 2..n \text{ div } 2\}$ 
    by (fastforce simp add: assms mult.commute)
  then show ?thesis
    by blast
qed

```

```

lemma odd_of_intvl_intvl:
  fixes  $S :: \text{nat set}$ 
  assumes  $S = \{m..n\} \cap \{i. \text{odd } i\}$ 
  shows  $\exists m' n'. S = \text{Suc `` } (\lambda i. i * 2) \text{ `` } \{m'..n'\}$ 
proof –
  have  $S = \text{Suc `` } (\{\text{if } n = 0 \text{ then } 1 \text{ else } m - 1..n - 1\} \cap \text{Collect even})$ 
    by (auto simp: assms image_def elim!: oddE)
  thus ?thesis
    by (metis even_of_intvl_intvl)
qed

```

```

lemma image_int_eq_image:
   $(\forall i \in S. f \ i \in T) \implies (f \text{ `` } S) \cap T = f \text{ `` } S$ 
   $(\forall i \in S. f \ i \notin T) \implies (f \text{ `` } S) \cap T = \{\}$ 
by auto

```

```

lemma braun_indices1_le:
   $i \in \text{braun\_indices } t \implies \text{Suc } 0 \leq i$ 
using braun_indices1_not_less_eq_eq by blast

```

```

lemma braun_if_braun_indices:  $\text{braun\_indices } t = \{1..size \ t\} \implies \text{braun } t$ 
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
then have insert (Suc 0) ((*) 2 ‘ braun_indices l ∪ Suc ‘ (*) 2 ‘
braun_indices r) ∩ {2..}
    = {Suc 0..Suc (size l + size r)} ∩ {2..}
by (metis Node.prem1 One_nat_def Suc_eq_plus1 Un_insert_left braun_indices.simps(2)
sup_bot_left tree.size(4))
then have eq: {2 .. size t} = (λi. i * 2) ‘ braun_indices l ∪ Suc ‘ (λi. i
* 2) ‘ braun_indices r
    (is ?R = ?S ∪ ?T)
by (simp add: t mult.commute Int_Un_distrib2 image_int_eq_image
braun_indices1_le)
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 size l = size r ∨ size l = size r + 1
    using t l r by atomize auto
with l r 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

```

39 Arrays via Braun Trees

```

theory Array_Braun
imports
    HOL-Library.Time_Functions
    Array_Specs
    Braun_Tree
begin

```

39.1 Array

fun *lookup1* :: 'a tree \Rightarrow nat \Rightarrow '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 \Rightarrow 'a \Rightarrow 'a tree \Rightarrow '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 \Rightarrow nat \Rightarrow 'a tree \Rightarrow '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 \Rightarrow 'a list **where**
list Leaf = [] |
list (Node *l x r*) = *x # splice* (*list l*) (*list r*)

39.1.1 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*: $\llbracket n < \text{size } xs + \text{size } ys; \text{size } ys \leq \text{size } xs; \text{size } xs \leq \text{size } ys + 1 \rrbracket$

$\implies \text{splice } xs \text{ } ys ! n = (\text{if even } n \text{ then } xs \text{ else } ys) ! (n \text{ div } 2)$

proof(*induction xs ys arbitrary: n rule: splice.induct*)

qed (*auto simp: nth_Cons' minus1_div2*)

lemma *div2_in_bounds*:

$\llbracket \text{braun } (\text{Node } l \text{ } x \text{ } r); n \in \{1.. \text{size}(\text{Node } l \text{ } x \text{ } r)\}; n > 1 \rrbracket \implies$
 $(\text{odd } n \longrightarrow n \text{ div } 2 \in \{1.. \text{size } r\}) \wedge (\text{even } n \longrightarrow n \text{ div } 2 \in \{1.. \text{size } l\})$

by *auto arith*

declare *upt_Suc*[*simp del*]

```

lookup1  lemma nth_list_lookup1:  $\llbracket \text{braun } t; i < \text{size } t \rrbracket \implies \text{list } t ! i =$ 
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.prem1, of i+1]
  by (auto simp: nth_splice minus1_div2 size_list)
qed

lemma list_eq_map_lookup1:  $\text{braun } t \implies \text{list } t = \text{map } (\text{lookup1 } t) [1..<\text{size}$ 
 $t + 1]$ 
by(auto simp add: list_eq_iff_nth_eq size_list nth_list_lookup1)

update1  lemma size_update1:  $\llbracket \text{braun } t; n \in \{1.. \text{size } t\} \rrbracket \implies \text{size}(\text{update1 } n$ 
 $x t) = \text{size } t$ 
proof(induction t arbitrary: n)
  case Leaf thus ?case by simp
next
  case Node thus ?case using div2_in_bounds[OF Node.prem1] by simp
qed

lemma braun_update1:  $\llbracket \text{braun } t; n \in \{1.. \text{size } t\} \rrbracket \implies \text{braun}(\text{update1 } n$ 
 $x t)$ 
proof(induction t arbitrary: n)
  case Leaf thus ?case by simp
next
  case Node thus ?case
  using div2_in_bounds[OF Node.prem1] by (simp add: size_update1)
qed

lemma lookup1_update1:  $\llbracket \text{braun } t; n \in \{1.. \text{size } t\} \rrbracket \implies$ 
 $\text{lookup1 } (\text{update1 } n x t) m = (\text{if } n=m \text{ then } x \text{ else } \text{lookup1 } t m)$ 
proof(induction t arbitrary: m n)
  case Leaf
  then show ?case by simp
next
  have aux:  $\llbracket \text{odd } n; \text{odd } m \rrbracket \implies n \text{ div } 2 = (m::\text{nat}) \text{ div } 2 \longleftrightarrow m=n$  for
  m n
  using odd_two_times_div_two_succ by fastforce
  case Node
  thus ?case using div2_in_bounds[OF Node.prem1] by (auto simp: aux)
qed

```

lemma *list_update1*: $\llbracket \text{braun } t; \ n \in \{1.. \text{size } t\} \rrbracket \implies \text{list}(\text{update1 } n \ x \ t)$
 $= (\text{list } t)[n-1 := x]$
by(*auto simp add: list_eq_map_lookup1 list_eq_iff_nth_eq lookup1_update1 size_update1 braun_update1*)

A second proof of $\llbracket \text{braun } ?t; \ ?n \in \{1.. \text{size } ?t\} \rrbracket \implies \text{list } (\text{update1 } ?n \ ?x \ ?t) = (\text{list } ?t)[?n - 1 := ?x]$:

lemma *diff1_eq_iff*: $n > 0 \implies n - \text{Suc } 0 = m \longleftrightarrow n = m+1$
by *arith*

lemma *list_update_splice*:
 $\llbracket n < \text{size } xs + \text{size } ys; \ \text{size } ys \leq \text{size } xs; \ \text{size } xs \leq \text{size } ys + 1 \rrbracket \implies$
 $(\text{splice } xs \ ys) [n := x] =$
 $(\text{if even } n \text{ then splice } (xs[n \text{ div } 2 := x]) \ ys \text{ else splice } xs \ (ys[n \text{ div } 2 := x]))$
by(*induction xs ys arbitrary: n rule: splice.induct*) (*auto split: nat.split*)

lemma *list_update2*: $\llbracket \text{braun } t; \ n \in \{1.. \text{size } t\} \rrbracket \implies \text{list}(\text{update1 } n \ x \ t)$
 $= (\text{list } t)[n-1 := x]$

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.prem]*

by(*auto simp: list_update_splice diff1_eq_iff size_list split: nat.split*)

qed

adds **lemma** *splice_last*: **shows**

$\text{size } ys \leq \text{size } xs \implies \text{splice } (xs @ [x]) \ ys = \text{splice } xs \ ys @ [x]$

and $\text{size } ys+1 \geq \text{size } xs \implies \text{splice } xs \ (ys @ [y]) = \text{splice } xs \ ys @ [y]$

by(*induction xs ys arbitrary: x y rule: splice.induct*) (*auto*)

lemma *list_add_hi*: $\text{braun } t \implies \text{list}(\text{update1 } (\text{Suc}(\text{size } t)) \ x \ t) = \text{list } t @ [x]$

by(*induction t*)(*auto simp: splice_last size_list*)

lemma *size_add_hi*: $\text{braun } t \implies m = \text{size } t \implies \text{size}(\text{update1 } (\text{Suc } m) \ x \ t) = \text{size } t + 1$

by(*induction t arbitrary: m*)(*auto*)

lemma *braun_add_hi*: $\text{braun } t \implies \text{braun}(\text{update1 } (\text{Suc}(\text{size } t)) \ x \ t)$

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

lemma *size_braun_adds*:

$\llbracket \text{braun } t; \ \text{size } t = n \rrbracket \implies \text{size}(\text{adds } xs \ n \ t) = \text{size } t + \text{length } xs \wedge \text{braun}$

```

(adds xs n t)
  by(induction xs arbitrary: t n)(auto simp: braun_add_hi size_add_hi)

lemma list_adds:  $\llbracket \text{braun } t; \text{size } t = n \rrbracket \implies \text{list}(\text{adds } xs \ n \ t) = \text{list } t @ xs$ 
  by(induction xs arbitrary: t n)(auto simp: size_braun_adds list_add_hi
  size_add_hi braun_add_hi)

```

39.1.2 Array Implementation

```

interpretation A: Array
  where lookup =  $\lambda(t,l) \ n. \text{lookup1 } t \ (n+1)$ 
    and update =  $\lambda n \ x \ (t,l). (\text{update1 } (n+1) \ x \ t, l)$ 
    and len =  $\lambda(t,l). l$ 
    and array =  $\lambda xs. (\text{adds } xs \ 0 \ \text{Leaf}, \text{length } xs)$ 
    and invar =  $\lambda(t,l). \text{braun } t \wedge l = \text{size } t$ 
    and list =  $\lambda(t,l). \text{list } t$ 
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

```

39.2 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))

```

39.2.1 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) = but-
last(list t)
by (induction t arbitrary: st) (auto simp: list_Nil_iff size_list butlast_splice)

```

```

lemma braun_del_hi: braun t ⇒ size t = st ⇒ braun(del_hi st t)
by (induction t arbitrary: st) (auto simp: list_del_hi simp flip: size_list)

```

39.2.2 Flexible Array Implementation

interpretation *AF*: *Array_Flex*

```

where lookup =  $\lambda(t,l)$  n. lookup1 t (n+1)
    and update =  $\lambda n x (t,l)$ . (update1 (n+1) x t, l)
    and len =  $\lambda(t,l)$ . l
    and array =  $\lambda xs$ . (adds xs 0 Leaf, length xs)
    and invar =  $\lambda(t,l)$ . braun t  $\wedge$  l = size t
    and list =  $\lambda(t,l)$ . list t
    and add_lo =  $\lambda x (t,l)$ . (add_lo x t, l+1)
    and del_lo =  $\lambda(t,l)$ . (del_lo t, l-1)
    and add_hi =  $\lambda x (t,l)$ . (update1 (Suc l) x t, l+1)
    and del_hi =  $\lambda(t,l)$ . (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

```

39.3 Faster

39.3.1 Size

fun diff :: 'a tree \Rightarrow nat \Rightarrow 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 \Rightarrow 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*: $braun\ t \implies size\ t : \{n, n + 1\} \implies diff\ t\ n = size\ t - n$
by (*induction t arbitrary: n*) *auto*

lemma *size_fast*: $braun\ t \implies size_fast\ t = size\ t$
by (*induction t*) (*auto simp add: diff*)

39.3.2 Initialization with 1 element

fun *braun_of_naive* :: '*a* \Rightarrow nat \Rightarrow '*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* \Rightarrow nat \Rightarrow '*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* \Rightarrow nat \Rightarrow '*a* tree **where**
braun_of *x* *n* = *fst* (*braun2_of* *x* *n*)

declare *braun2_of.simps* [*simp del*]

lemma *braun2_of_size_braun*: $braun2_of\ x\ n = (s,t) \implies size\ s = n \wedge size\ t = n+1 \wedge braun\ s \wedge braun\ t$

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*:

$braun2_of\ x\ n = (s,t) \implies list\ s = replicate\ n\ x \wedge list\ t = replicate\ (n+1)\ x$

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 eqfst_iff braun2_of_size_braun)

```

```

corollary list_braun_of: list(braun_of x n) = replicate n x
  unfolding braun_of_def by (metis eqfst_iff braun2_of_replicate)

```

39.3.3 Proof Infrastructure

Originally due to Thomas Sewell.

```

take_nths fun take_nths :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  take_nths i k [] = [] |
  take_nths i k (x # xs) = (if i = 0 then x # take_nths (2k - 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 = nthxs xs ( $\bigcup n. \{n * 2^k + i\}$ )
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  $\bigwedge x n. \text{Suc } x = n * 2^k \implies \exists xa. x = \text{Suc } xa * 2^k - \text{Suc } 0$ 
      by (metis diff_Suc_Suc diff_zero mult_eq_0_iff not0_implies_Suc)
    then have ( $\bigcup n. \{(n+1) * 2^k - 1\}$ ) = {m.  $\exists n. \text{Suc } m = n * 2^k$ }
      by (auto simp del: mult_Suc)
    thus ?thesis by (simp add: Cons.IH ac_simps nthxs_Cons)
  next
    assume [arith]: i  $\neq$  0
    have  $\bigwedge x n. \text{Suc } x = n * 2^k + i \implies \exists xa. x = xa * 2^k + i - \text{Suc } 0$ 
      by (metis diff_Suc_Suc diff_zero)
    then have ( $\bigcup n. \{n * 2^k + i - 1\}$ ) = {m.  $\exists n. \text{Suc } m = n * 2^k + i$ }
      by auto
    thus ?thesis by (simp add: Cons.IH nthxs_Cons)
  qed

```

qed
qed

lemma *take_nth_drop*:
 $take_nth\ i\ k\ (drop\ j\ xs) = take_nth\ (i + j)\ k\ xs$
by (*induct xs arbitrary: i j; simp add: drop_Cons split: nat.split*)

lemma *take_nth_00*:
 $take_nth\ 0\ 0\ xs = xs$
by (*induct xs; simp*)

lemma *splice_take_nth*:
 $splice\ (take_nth\ 0\ (Suc\ 0)\ xs)\ (take_nth\ (Suc\ 0)\ (Suc\ 0)\ xs) = xs$
by (*induct xs; simp*)

lemma *take_nth_take_nth*:
 $take_nth\ i\ m\ (take_nth\ j\ n\ xs) = take_nth\ ((i * 2^n) + j)\ (m + n)\ xs$
by (*induct xs arbitrary: i j; simp add: algebra_simps power_add*)

lemma *take_nth_empty*:
 $(take_nth\ i\ k\ xs = []) = (length\ xs \leq i)$
by (*induction xs arbitrary: i k auto*)

lemma *hd_take_nth*:
 $i < length\ xs \implies hd(take_nth\ i\ k\ xs) = xs\ !\ i$
by (*induction xs arbitrary: i k auto*)

lemma *take_nth_01_splice*:
 $\llbracket length\ xs = length\ ys \vee length\ xs = length\ ys + 1 \rrbracket \implies$
 $take_nth\ 0\ (Suc\ 0)\ (splice\ xs\ ys) = xs \wedge$
 $take_nth\ (Suc\ 0)\ (Suc\ 0)\ (splice\ xs\ ys) = ys$
by (*induct xs arbitrary: ys; case_tac ys; simp*)

lemma *length_take_nth_00*:
 $length\ (take_nth\ 0\ (Suc\ 0)\ xs) = length\ (take_nth\ (Suc\ 0)\ (Suc\ 0)\ xs)$
 \vee
 $length\ (take_nth\ 0\ (Suc\ 0)\ xs) = length\ (take_nth\ (Suc\ 0)\ (Suc\ 0)\ xs)$
 $+ 1$
by (*induct xs auto*)

braun_list **fun** *braun_list* :: 'a tree \Rightarrow 'a list \Rightarrow bool **where**
 $braun_list\ Leaf\ xs = (xs = [])$ |
 $braun_list\ (Node\ l\ x\ r)\ xs = (xs \neq [] \wedge x = hd\ xs \wedge$

```

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

```

39.3.4 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 (2k) xs;
        zs = drop (2k) xs;
        ts = brauns (k+1) zs
    in nodes ts ys (drop (2k) ts))

```

```

declare brauns.simps[simp del]

```

```

definition brauns1 :: 'a list ⇒ 'a tree where
  brauns1 xs = (if xs = [] then Leaf else brauns 0 xs ! 0)

```

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)  $\implies$  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  $\neq$  [] using less.prem by auto
  let ?zs = drop (2^k) xs
  let ?ts = brauns (Suc k) ?zs
  from less.hyps[of ?zs _ Suc k]
  have IH:  $\llbracket j = i + 2^k; i < \min (\text{length } ?zs) (2^{k+1}) \rrbracket \implies$ 
    braun_list (?ts ! i) (take_nths j (Suc k) xs) for i j
  using  $\langle xs \neq [] \rangle$  by (simp add: take_nths_drop)
  show ?case
  using less.prem
  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)  $\wedge$  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 time_fun_0 ( $\wedge$ )

time_fun nodes

lemma T_nodes: T_nodes ls xs rs = length xs + 1
by(induction ls xs rs rule: T_nodes.induct) auto

time_fun brauns

lemma T_brauns_simple: T_brauns k xs = (if xs = [] then 0 else

```

$3 * (\min (2^k) (\text{length } xs) + 1) + (\min (2^k) (\text{length } xs - 2^k) + 1)$
 $+ T_brauns (k+1) (\text{drop } (2^k) xs)) + 1$
by(*simp add: T_nodes T_take T_drop length_brauns min_def*)

theorem *T_brauns_ub*:

$T_brauns\ k\ xs \leq 9 * (\text{length } xs + 1)$

proof (*induction xs arbitrary: k rule: measure_induct_rule*[**where** $f = \text{length}$])

case (*less xs*)

show *?case*

proof *cases*

assume $xs = []$

thus *?thesis* **by**(*simp*)

next

assume $xs \neq []$

let $?n = \text{length } xs$ **let** $?zs = \text{drop } (2^k) xs$

have $*$: $?n - 2^k + 1 \leq ?n$

using $\langle xs \neq [] \rangle$ *less_eq_Suc_le* **by** *fastforce*

have $T_brauns\ k\ xs =$

$3 * (\min (2^k) ?n + 1) + (\min (2^k) (?n - 2^k) + 1) + T_brauns$
 $(k+1) ?zs + 1$

unfolding *T_brauns_simple*[*of k xs*] **using** $\langle xs \neq [] \rangle$ **by**(*simp del: T_brauns.simps*)

also have $\dots \leq 4 * \min (2^k) ?n + T_brauns (k+1) ?zs + 5$

by(*simp add: min_def*)

also have $\dots \leq 4 * \min (2^k) ?n + 9 * (\text{length } ?zs + 1) + 5$

using *less*[*of ?zs k+1*] $\langle xs \neq [] \rangle$

by (*simp del: T_brauns.simps*)

also have $\dots = 4 * \min (2^k) ?n + 9 * (?n - 2^k + 1) + 5$

by(*simp*)

also have $\dots = 4 * \min (2^k) ?n + 4 * (?n - 2^k) + 5 * (?n - 2^k$
 $+ 1) + 9$

by(*simp*)

also have $\dots = 4 * ?n + 5 * (?n - 2^k + 1) + 9$

by(*simp*)

also have $\dots \leq 4 * ?n + 5 * ?n + 9$

using $*$ **by**(*simp*)

also have $\dots = 9 * (?n + 1)$

by (*simp add: Suc_leI*)

finally show *?thesis* .

qed

qed

39.3.5 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  $\Rightarrow$  'a list where
  list_fast_rec ts = (let us = filter ( $\lambda t. t \neq \text{Leaf}$ ) 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  $\neq [] \implies \text{Leaf} \notin \text{set } ts \implies$ 
  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  $\neq [] \implies us = \text{filter } (\lambda t. t \neq \langle \rangle) ts \implies$ 
  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
  by (relation measure (sum_list o map size); simp add: list_fast_rec_term)

declare list_fast_rec.simps[simp del]

definition list_fast :: 'a tree  $\Rightarrow$  'a list where
  list_fast t = list_fast_rec [t]

definition filter_not_Leaf = filter ( $\lambda t. t \neq \text{Leaf}$ )

definition map_left = map left
definition map_right = map right
definition map_value = map value

definition T_filter_not_Leaf ts = length ts + 1
definition T_map_left ts = length ts + 1
definition T_map_right ts = length ts + 1

```

definition $T_map_value\ ts = length\ ts + 1$

lemmas $defs = filter_not_Leaf_def\ map_left_def\ map_right_def\ map_value_def$
 $T_filter_not_Leaf_def\ T_map_value_def\ T_map_left_def\ T_map_right_def$

lemma $list_fast_rec_simp$:

$list_fast_rec\ ts = (let\ us = filter_not_Leaf\ ts\ in$
 $\quad if\ us = []\ then\ []\ else$
 $\quad map_value\ us\ @\ list_fast_rec\ (map_left\ us\ @\ map_right\ us))$

unfolding $defs\ list_fast_rec.simps[of\ ts]\ by(rule\ refl)$

time_function $list_fast_rec\ equations\ list_fast_rec_simp$

termination

by $(relation\ measure\ (sum_list\ o\ map\ size); simp\ add: list_fast_rec_term\ defs)$

declare $T_list_fast_rec.simps[simp\ del]$

Functional Correctness **lemma** $list_fast_rec_all_Leaf$:

$\forall t \in set\ ts. t = Leaf \implies list_fast_rec\ ts = []$
by $(simp\ add: filter_empty_conv\ list_fast_rec.simps)$

lemma $take_nth_eq_single$:

$length\ xs - i < 2^n \implies take_nth\ i\ n\ xs = take\ 1\ (drop\ i\ xs)$
by $(induction\ xs\ arbitrary: i\ n; simp\ add: drop_Cons')$

lemma $braun_list_Nil$:

$braun_list\ t\ [] = (t = Leaf)$
by $(cases\ t; simp)$

lemma $braun_list_not_Nil$: $xs \neq [] \implies$

$braun_list\ t\ xs =$
 $(\exists l\ x\ r. t = Node\ l\ x\ r \wedge x = hd\ xs \wedge$
 $\quad braun_list\ l\ (take_nth\ 1\ 1\ xs) \wedge$
 $\quad braun_list\ r\ (take_nth\ 2\ 1\ xs))$

by $(cases\ t; simp)$

theorem $list_fast_rec_correct$:

$[length\ ts = 2^k; \forall i < 2^k. braun_list\ (ts\ !\ i)\ (take_nth\ i\ k\ xs)]$
 $\implies list_fast_rec\ ts = xs$

proof $(induct\ xs\ arbitrary: k\ ts\ rule: measure_induct_rule[where\ f=length])$

```

case (less xs)
show ?case
proof (cases length xs < 2 ^ k)
  case True
  from less.prems True have filter:
     $\exists n. ts = \text{map } (\lambda x. \text{Node Leaf } x \text{ Leaf}) \text{ } xs @ \text{replicate } n \text{ Leaf}$ 
    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 *:
     $\forall i < 2 ^ k. ts ! i \neq \text{Leaf}$ 
     $\wedge \text{value } (ts ! i) = xs ! i$ 
     $\wedge \text{braun\_list } (\text{left } (ts ! i)) (\text{take\_nth} s \ (i + 2 ^ k) \ (\text{Suc } k) \ xs)$ 
     $\wedge (\forall ys. ys = \text{take\_nth} s \ (i + 2 * 2 ^ k) \ (\text{Suc } k) \ xs$ 
       $\longrightarrow \text{braun\_list } (\text{right } (ts ! i)) \ ys)$ 
    by (auto simp: take_nth_empty hd_take_nth braun_list_not_Nil
take_nth_take_nth
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_nth_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  $\implies$  list_fast t = list t
  by (simp add: list_fast_def take_nth_00 braun_list_eq list_fast_rec_correct[where
k=0])

```

Running Time Analysis lemma *sum_tree_list_children*: $\forall t \in \text{set } ts. t \neq \text{Leaf} \implies$
 $(\sum t \leftarrow ts. k * \text{size } t) = (\sum t \leftarrow \text{map left } ts @ \text{map right } ts. k * \text{size } t) +$


```

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. 14 * size t + 1) ts) + 2
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: defs sum_list_Suc)
  next
    assume ?us ≠ []
    let ?children = map left ?us @ map right ?us
    have 1: 1 ≤ length ?us
      using ⟨?us ≠ []⟩ linorder_not_less by auto
    have T_list_fast_rec ts = T_list_fast_rec ?children + 5 * length ?us
      + length ts + 7
      using ⟨?us ≠ []⟩ T_list_fast_rec.simps[of ts] by(simp add: defs
T_append)
    also have ... ≤ (∑ t←?children. 14 * size t + 1) + 5 * length ?us +
length ts + 9
      using less[of ?children] list_fast_rec_term[of ?us] ⟨?us ≠ []⟩
      by (simp)
    also have ... = (∑ t←?children. 14 * size t) + 7 * length ?us + length
ts + 9
      by(simp add: sum_list_Suc o_def)
    also have ... ≤ (∑ t←?children. 14 * size t) + 14 * length ?us +
length ts + 2
      using 1 by(simp add: sum_list_Suc o_def)
    also have ... = (∑ t←?us. 14 * size t) + length ts + 2
      by(simp add: sum_tree_list_children)
    also have ... ≤ (∑ t←ts. 14 * size t) + length ts + 2
      by(simp add: sum_list_filter_le_nat)
    also have ... = (∑ t←ts. 14 * size t + 1) + 2
      by(simp add: sum_list_Suc)
    finally show ?case .
  qed
qed
end

```

40 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  $\Rightarrow$  'a trie option
```

```
definition empty :: 'a trie where
[simp]: empty = Nd False ( $\lambda$ _. None)
```

```
fun isin :: 'a trie  $\Rightarrow$  'a list  $\Rightarrow$  bool where
isin (Nd b m) [] = b |
isin (Nd b m) (k # xs) = (case m k of None  $\Rightarrow$  False | Some t  $\Rightarrow$  isin t xs)
```

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

```
fun delete :: 'a list  $\Rightarrow$  'a trie  $\Rightarrow$  'a trie where
delete [] (Nd b m) = Nd False m |
delete (x#xs) (Nd b m) = Nd b
  (case m x of
    None  $\Rightarrow$  m |
    Some t  $\Rightarrow$  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
    []  $\Rightarrow$  b |
    x # ys  $\Rightarrow$  (case m x of None  $\Rightarrow$  False | Some t  $\Rightarrow$  isin t ys))
by(cases xs)auto
```

```
definition set_trie :: 'a trie  $\Rightarrow$  'a list set where
[simp]: set_trie t = {xs. isin t xs}
```

```
lemma isin_set_trie: isin t xs = (xs  $\in$  set_trie t)
by simp
```

```

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

lemma set_trie_delete: set_trie (delete xs t) = set_trie 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_trie and invar =  $\lambda \_.$  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_trie)
next
    case 3 show ?case by(rule set_trie_insert)
next
    case 4 show ?case by(rule set_trie_delete)
qed (rule TrueI)+

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  $\Rightarrow$  'a * 'a  $\Rightarrow$  'a where
    sel2 b (a1,a2) = (if b then a2 else a1)

fun mod2 :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  bool  $\Rightarrow$  'a * 'a  $\Rightarrow$  '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* \Rightarrow *bool list* \Rightarrow *bool* **where**
isin *Lf* *ks* = *False* |
isin (*Nd b lr*) *ks* =
 (case *ks* of
 [] \Rightarrow *b* |
 k#ks \Rightarrow *isin* (*sel2 k lr*) *ks*)

fun *insert* :: *bool list* \Rightarrow *trie* \Rightarrow *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* \vee *isin t ys*)
proof (*induction xs t arbitrary: ys rule: insert.induct*)
qed (*auto split: list.splits if_splits*)

A simple implementation of delete; does not shrink the trie!

fun *delete0* :: *bool list* \Rightarrow *trie* \Rightarrow *trie* **where**
delete0 ks Lf = *Lf* |
delete0 ks (Nd b lr) =
 (case *ks* of
 [] \Rightarrow *Nd False lr* |
 k#ks' \Rightarrow *Nd b (mod2 (delete0 ks') k lr)*)

lemma *isin_delete0*: *isin* (*delete0 as t*) *bs* = (*as* \neq *bs* \wedge *isin t bs*)
proof (*induction as t arbitrary: bs rule: delete0.induct*)
qed (*auto split: list.splits if_splits*)

Now deletion with shrinking:

fun *node* :: *bool* \Rightarrow *trie* * *trie* \Rightarrow *trie* **where**
node b lr = (*if* \neg *b* \wedge *lr* = (*Lf,Lf*) *then Lf* *else Nd b lr*)

fun *delete* :: *bool list* \Rightarrow *trie* \Rightarrow *trie* **where**
delete ks Lf = *Lf* |
delete ks (Nd b lr) =
 (case *ks* of
 [] \Rightarrow *node False lr* |
 k#ks' \Rightarrow *node b (mod2 (delete ks') k lr)*)

lemma *isin_delete*: *isin* (*delete xs t*) *ys* = (*xs* \neq *ys* \wedge *isin t ys*)

```

apply(induction xs t arbitrary: ys rule: delete.induct)
  apply (auto split: list.splits if_splits)
  apply (metis isin.simps(1))+
done

definition set_trie :: trie  $\Rightarrow$  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  $\in$  set_trie t)
  by(simp add: set_trie_def)

lemma set_trie_insert: set_trie(insert xs t) = set_trie t  $\cup$  {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)

  Invariant: tries are fully shrunk:

fun invar where
  invar Lf = True |
  invar (Nd b (l,r)) = (invar l  $\wedge$  invar r  $\wedge$  (l = Lf  $\wedge$  r = Lf  $\longrightarrow$  b))

lemma insert_Lf: insert xs t  $\neq$  Lf
  using insert.elims by blast

lemma invar_insert: invar t  $\implies$  invar(insert xs t)
proof(induction xs t rule: insert.induct)
  case 1 thus ?case by simp
next
  case (2 b lr)
  thus ?case by(cases lr; simp)
next
  case (3 k ks)
  thus ?case by(simp; cases ks; auto)
next
  case (4 k ks b lr)
  then show ?case by(cases lr; auto simp: insert_Lf)
qed

lemma invar_delete: invar t  $\implies$  invar(delete xs t)
proof(induction t arbitrary: xs)

```

```

    case Lf thus ?case by simp
next
  case (Nd b lr)
  thus ?case by (cases lr) (auto split: list.split)
qed

```

interpretation *S*: *Set*

```

  where empty = empty and isin = isin and insert = insert and delete
= delete
  and set = set_trie and invar = invar
  unfolding Set_def
  by (smt (verit, best) Tries_Binary.empty_def invar.simps(1) invar_delete
invar_insert set_trie_delete set_trie_empty set_trie_insert set_trie_isin)

```

41.2 Patricia Trie

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

Fully shrunk:

fun *invarP* **where**

```

  invarP LfP = True |
  invarP (NdP ps b (l,r)) = (invarP l ∧ invarP r ∧ (l = LfP ∨ r = LfP
→ b))

```

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 *lcp* :: 'a *list* ⇒ 'a *list* ⇒ 'a *list* × 'a *list* × 'a *list* **where**

```

  lcp [] ys = ([], [], ys) |
  lcp xs [] = ([], xs, []) |
  lcp (x#xs) (y#ys) =
  (if x≠y then ([], x#xs, y#ys)
  else let (ps, xs', ys') = lcp xs ys in (x#ps, xs', ys'))

```

lemma *mod2_cong*[*fundef_cong*]:

$$\llbracket lr = lr'; k = k'; \bigwedge a \ b. \ lr' = (a, b) \implies f(a) = f'(a) ; \bigwedge a \ b. \ lr' = (a, b) \implies f(b) = f'(b) \rrbracket$$

$$\implies \text{mod2 } f \ k \ lr = \text{mod2 } f' \ k' \ lr'$$
by(cases *lr*, cases *lr'*, *auto*)

fun *insertP* :: *bool list* \Rightarrow *trieP* \Rightarrow *trieP* **where**
insertP *ks* *LfP* = *NdP* *ks* *True* (*LfP*, *LfP*) |
insertP *ks* (*NdP* *ps* *b* *lr*) =
(case *lcp* *ks* *ps* of
(*qs*, *k#ks'*, *p#ps'*) \Rightarrow
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'*, []) \Rightarrow
NdP *ps* *b* (mod2 (*insertP* *ks'*) *k* *lr*) |
(*qs*, [], *p#ps'*) \Rightarrow
let *t* = *NdP* *ps'* *b* *lr* in
NdP *qs* *True* (if *p* then (*LfP*, *t*) else (*t*, *LfP*)) |
(*qs*, [], []) \Rightarrow *NdP* *ps* *True* *lr*)

Smart constructor that shrinks:

definition *nodeP* :: *bool list* \Rightarrow *bool* \Rightarrow *trieP* * *trieP* \Rightarrow *trieP* **where**
nodeP *ps* *b* *lr* =
(if *b* then *NdP* *ps* *b* *lr*
else case *lr* of
(*LfP*, *LfP*) \Rightarrow *LfP* |
(*LfP*, *NdP* *ks* *b* *lr*) \Rightarrow *NdP* (*ps* @ *True* # *ks*) *b* *lr* |
(*NdP* *ks* *b* *lr*, *LfP*) \Rightarrow *NdP* (*ps* @ *False* # *ks*) *b* *lr* |
_ \Rightarrow *NdP* *ps* *b* *lr*)

fun *deleteP* :: *bool list* \Rightarrow *trieP* \Rightarrow *trieP* **where**
deleteP *ks* *LfP* = *LfP* |
deleteP *ks* (*NdP* *ps* *b* *lr*) =
(case *lcp* *ks* *ps* of
(_, _, _#_) \Rightarrow *NdP* *ps* *b* *lr* |
(_, *k#ks'*, []) \Rightarrow *nodeP* *ps* *b* (mod2 (*deleteP* *ks'*) *k* *lr*) |
(_, [], []) \Rightarrow *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* \Rightarrow *trie* \Rightarrow *trie* **where**
prefix_trie [] *t* = *t* |
prefix_trie (*k#ks*) *t* =

(let $t' = \text{prefix_trie } ks \ t \text{ in } Nd \text{ False (if } k \text{ then (Lf,t') else (t',Lf))})$)

fun $\text{abs_trieP} :: \text{trieP} \Rightarrow \text{trie}$ **where**
 $\text{abs_trieP } LfP = Lf \mid$
 $\text{abs_trieP } (NdP \ ps \ b \ (l,r)) = \text{prefix_trie } ps \ (Nd \ b \ (\text{abs_trieP } l, \text{abs_trieP } r))$

Correctness of isinP :

lemma isin_prefix_trie :
 $\text{isin } (\text{prefix_trie } ps \ t) \ ks$
 $= (ps = \text{take } (\text{length } ps) \ ks \wedge \text{isin } t \ (\text{drop } (\text{length } ps) \ ks))$
by ($\text{induction } ps \text{ arbitrary: } ks$) ($\text{auto split: list.split}$)

lemma abs_trieP_isinP :
 $\text{isinP } t \ ks = \text{isin } (\text{abs_trieP } t) \ ks$
proof ($\text{induction } t \text{ arbitrary: } ks \text{ rule: abs_trieP.induct}$)
qed ($\text{auto simp: isin_prefix_trie split: list.split}$)

Correctness of insertP :

lemma prefix_trie_Lfs : $\text{prefix_trie } ks \ (Nd \ \text{True } (Lf,Lf)) = \text{insert } ks \ Lf$
by ($\text{induction } ks$) auto

lemma $\text{insert_prefix_trie_same}$:
 $\text{insert } ps \ (\text{prefix_trie } ps \ (Nd \ b \ lr)) = \text{prefix_trie } ps \ (Nd \ \text{True } lr)$
by ($\text{induction } ps$) auto

lemma insert_append : $\text{insert } (ks \ @ \ ks') \ (\text{prefix_trie } ks \ t) = \text{prefix_trie } ks \ (\text{insert } ks' \ t)$
by ($\text{induction } ks$) auto

lemma $\text{prefix_trie_append}$: $\text{prefix_trie } (ps \ @ \ qs) \ t = \text{prefix_trie } ps \ (\text{prefix_trie } qs \ t)$
by ($\text{induction } ps$) auto

lemma lcp_if : $\text{lcp } ks \ ps = (qs, ks', ps') \implies$
 $ks = qs \ @ \ ks' \wedge ps = qs \ @ \ ps' \wedge (ks' \neq [] \wedge ps' \neq [] \longrightarrow \text{hd } ks' \neq \text{hd } ps')$
proof ($\text{induction } ks \ ps \text{ arbitrary: } qs \ ks' \ ps' \text{ rule: lcp.induct}$)
qed ($\text{auto split: prod.splits if_splits}$)

lemma abs_trieP_insertP :
 $\text{abs_trieP } (\text{insertP } ks \ t) = \text{insert } ks \ (\text{abs_trieP } t)$
proof ($\text{induction } t \text{ arbitrary: } ks$)
qed ($\text{auto simp: prefix_trie_Lfs insert_prefix_trie_same insert_append prefix_trie_append}$)

dest!: *lcp_if split*: *list.split prod.split if_splits*)

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 *nodeP_LfP2*: *nodeP xs False (LfP, LfP) = LfP*
by(*simp add: nodeP_def*)

Some non-inductive aux. lemmas:

lemma *abs_trieP_nodeP*: *a* \neq *LfP* \vee *b* \neq *LfP* \implies
abs_trieP (nodeP xs f (a, b)) = prefix_trie xs (Nd f (abs_trieP a,
abs_trieP b))
by(*auto simp add: nodeP_def prefix_trie_append split: trieP.split*)

lemma *nodeP_True*: *nodeP ps True lr = NdP ps True lr*
by(*simp add: nodeP_def*)

lemma *delete_abs_trieP*:
delete ks (abs_trieP t) = abs_trieP (deleteP ks t)
proof (*induction t arbitrary: ks*)
qed (*auto simp: delete_prefix_trie delete_append_prefix_trie*
prefix_trie_append prefix_trie_Lf abs_trieP_Lf nodeP_LfP2 abs_trieP_nodeP
nodeP_True
dest!: *lcp_if split: if_splits list.split prod.split*)

Invariant preservation:

lemma *insertP_LfP*: *insertP xs t* \neq *LfP*
by(*cases t*)(*auto split: prod.split list.split*)

lemma *invarP_insertP*: *invarP t* \implies *invarP (insertP xs t)*

```

proof(induction t arbitrary: xs)
  case LfP thus ?case by simp
next
  case (NdP bs b lr)
  then show ?case
    by(cases lr)(auto simp: insertP_LfP split: prod.split list.split)
qed

```

```

lemma invarP_nodeP:  $\llbracket \text{invarP } t1; \text{invarP } t2 \rrbracket \implies \text{invarP } (\text{nodeP } xs \ b \ (t1, \ t2))$ 
  by (auto simp add: nodeP_def split: trieP.split)

```

```

lemma invarP_deleteP: invarP t  $\implies \text{invarP}(\text{deleteP } xs \ t)$ 
proof(induction t arbitrary: xs)
  case LfP thus ?case by simp
next
  case (NdP ks b lr)
  thus ?case by(cases lr)(auto simp: invarP_nodeP split: prod.split list.split)
qed

```

The overall correctness proof. Simply composes correctness lemmas.

```

definition set_trieP :: trieP  $\Rightarrow$  bool list set where
  set_trieP = set_trie o abs_trieP

```

```

lemma isinP_set_trieP: isinP t xs = (xs  $\in$  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  $\cup$  {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 = invarP
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)
next
  case 5 thus ?case by(simp)
next
  case 6 thus ?case by(rule invarP_insertP)
next
  case 7 thus ?case by(rule invarP_deleteP)
qed

end

```

42 Ternary Tries

```

theory Trie_Ternary
imports
  Tree_Map
  Trie_Fun
begin

```

An implementation of tries for an arbitrary alphabet $'a$ where the mapping from an element of type $'a$ to the sub-trie is implemented by an (unbalanced) binary search tree. In principle, other search trees (e.g. red-black trees) work just as well, with some small adjustments (Exercise!).

This is an implementation of the “ternary search trees” by Bentley and Sedgewick [SODA 1997, Dr. Dobbs 1998]. The name derives from the fact that a node in the BST can now be drawn to have 3 children, where the middle child is the sub-trie that the node maps its key to. Hence the name *trie3*.

Example from https://en.wikipedia.org/wiki/Ternary_search_tree#Description:

c / | a u h | | | t. t e. u / / | / | s. p. e. i. s.

Characters with a dot are final. Thus the tree represents the set of strings "cute", "cup", "at", "as", "he", "us" and "i".

```

datatype 'a trie3 = Nd3 bool ('a * 'a trie3) tree

```

The development below works almost verbatim for any search tree implementation, eg *RBT_Map*, and not just *Tree_Map*, except for the termination lemma *lookup_size*.

```

term size_tree

```

```

lemma lookup_size[termination_simp]:

```

```

  fixes t :: ('a::linorder * 'a trie3) tree

```

```

  shows lookup t a = Some b  $\implies$  size b < Suc (size_tree ( $\lambda$ ab. Suc (size (snd( ab)))) t)

```

```

  by (induction t a rule: lookup.induct)(auto split: if_splits)

```

definition *empty3* :: 'a trie3 **where**

[simp]: *empty3* = Nd3 False Leaf

fun *isin3* :: ('a::linorder) trie3 \Rightarrow 'a list \Rightarrow bool **where**

isin3 (Nd3 b m) [] = b |

isin3 (Nd3 b m) (x # xs) = (case lookup m x of None \Rightarrow False | Some t \Rightarrow *isin3* t xs)

fun *insert3* :: ('a::linorder) list \Rightarrow 'a trie3 \Rightarrow 'a trie3 **where**

insert3 [] (Nd3 b m) = Nd3 True m |

insert3 (x#xs) (Nd3 b m) =

Nd3 b (update x (*insert3* xs (case lookup m x of None \Rightarrow *empty3* | Some t \Rightarrow t)) m)

fun *delete3* :: ('a::linorder) list \Rightarrow 'a trie3 \Rightarrow 'a trie3 **where**

delete3 [] (Nd3 b m) = Nd3 False m |

delete3 (x#xs) (Nd3 b m) = Nd3 b

(case lookup m x of

None \Rightarrow m |

Some t \Rightarrow update x (*delete3* xs t) m)

42.1 Correctness

Proof by stepwise refinement. First *abs3tract* to type 'a trie.

fun *abs3* :: 'a::linorder trie3 \Rightarrow 'a trie **where**

abs3 (Nd3 b t) = Nd b ($\lambda a.$ map_option *abs3* (lookup t a))

fun *invar3* :: ('a::linorder)trie3 \Rightarrow bool **where**

invar3 (Nd3 b m) = (M.invar m \wedge ($\forall a t.$ lookup m a = Some t \longrightarrow *invar3* t))

lemma *isin_abs3*: *isin3* t xs = *isin* (*abs3* t) xs

by (induction t xs rule: *isin3.induct*)(auto split: option.split)

lemma *abs3_insert3*: *invar3* t \Longrightarrow *abs3*(*insert3* xs t) = *insert* xs (*abs3* t)

proof (induction xs t rule: *insert3.induct*)

qed (auto simp: M.map_specs Tree_Set.empty_def[symmetric] split: option.split)

lemma *abs3_delete3*: *invar3* t \Longrightarrow *abs3*(*delete3* xs t) = *delete* xs (*abs3* t)

by (induction xs t rule: *delete3.induct*)(auto simp: M.map_specs split: option.split)

```

lemma invar3_insert3: invar3 t  $\implies$  invar3 (insert3 xs t)
proof (induction xs t rule: insert3.induct)
qed (auto simp: M.map_specs simp flip: Tree_Set.empty_def split: option.split)

```

```

lemma invar3_delete3: invar3 t  $\implies$  invar3 (delete3 xs t)
  by (induction xs t rule: delete3.induct)(auto simp: M.map_specs split: option.split)

```

Overall correctness w.r.t. the *Set* ADT:

```

interpretation S2: Set
where empty = empty3 and isin = isin3 and insert = insert3 and delete
= delete3
and set = set_trie o abs3 and invar = invar3
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_abs3)
next
  case 3 thus ?case by (simp add: set_trie_insert abs3_insert3 del: set_trie_def)
next
  case 4 thus ?case by (simp add: set_trie_delete abs3_delete3 del: set_trie_def)
next
  case 5 thus ?case by (simp add: M.map_specs Tree_Set.empty_def[symmetric])
next
  case 6 thus ?case by (simp add: invar3_insert3)
next
  case 7 thus ?case by (simp add: invar3_delete3)
qed

end

```

43 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  $\implies$  list (enq x q) = list q @ [x]
assumes list_deq:    invar q  $\implies$  list (deq q) = tl (list q)
assumes list_first:  invar q  $\implies$   $\neg$  list q = []  $\implies$  first q = hd (list q)
assumes list_is_empty: invar q  $\implies$  is_empty q = (list q = [])
assumes invar_empty: invar empty
assumes invar_enq:   invar q  $\implies$  invar (enq x q)
assumes invar_deq:   invar q  $\implies$  invar (deq q)

end

```

44 Queue Implementation via 2 Lists

```

theory Queue_2Lists
imports
  Queue_Spec
  HOL-Library.Time_Functions
begin

  Definitions:

  type_synonym 'a queue = 'a list  $\times$  'a list

  fun norm :: 'a queue  $\Rightarrow$  'a queue where
    norm (fs,rs) = (if fs = [] then (itrev rs [], []) else (fs,rs))

  fun enq :: 'a  $\Rightarrow$  'a queue  $\Rightarrow$  'a queue where
    enq a (fs,rs) = norm (fs, a # rs)

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

  fun first :: 'a queue  $\Rightarrow$  'a where
    first (a # fs,rs) = a

  fun is_empty :: 'a queue  $\Rightarrow$  bool where
    is_empty (fs,rs) = (fs = [])

  fun list :: 'a queue  $\Rightarrow$  'a list where
    list (fs,rs) = fs @ rev rs

```

```
fun invar :: 'a queue  $\Rightarrow$  bool where
invar (fs,rs) = (fs = []  $\longrightarrow$  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:

```
time_fun norm
time_fun enq
time_fun deq
```

Amortized running times:

```
fun  $\Phi$  :: 'a queue  $\Rightarrow$  nat where
 $\Phi$ (fs,rs) = length rs
```

```
lemma a_enq:  $T\_enq\ a\ (fs,rs) + \Phi(enq\ a\ (fs,rs)) - \Phi(fs,rs) \leq 2$ 
by(auto simp: T_itrev)
```

```
lemma a_deq:  $T\_deq\ (fs,rs) + \Phi(deq\ (fs,rs)) - \Phi(fs,rs) \leq 1$ 
by(auto simp: T_itrev T_tl)
```

```
end
```

45 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  $\Rightarrow$  bool
and insert :: 'a::linorder  $\Rightarrow$  'q  $\Rightarrow$  'q
and get_min :: 'q  $\Rightarrow$  'a
and del_min :: 'q  $\Rightarrow$  'q
and invar :: 'q  $\Rightarrow$  bool
and mset :: 'q  $\Rightarrow$  'a multiset
assumes mset_empty: mset empty = {#}
and is_empty: invar q  $\Longrightarrow$  is_empty q = (mset q = {#})
and mset_insert: invar q  $\Longrightarrow$  mset (insert x q) = mset q + {#x#}
and mset_del_min: invar q  $\Longrightarrow$  mset q  $\neq$  {#}  $\Longrightarrow$ 
  mset (del_min q) = mset q - {# get_min q #}
and mset_get_min: invar q  $\Longrightarrow$  mset q  $\neq$  {#}  $\Longrightarrow$  get_min q = Min_mset
  (mset q)
and invar_empty: invar empty
and invar_insert: invar q  $\Longrightarrow$  invar (insert x q)
and invar_del_min: invar q  $\Longrightarrow$  mset q  $\neq$  {#}  $\Longrightarrow$  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  $\Rightarrow$  'q  $\Rightarrow$  'q
assumes mset_merge:  $\llbracket$  invar q1; invar q2  $\rrbracket \Longrightarrow$  mset (merge q1 q2) =
  mset q1 + mset q2
and invar_merge:  $\llbracket$  invar q1; invar q2  $\rrbracket \Longrightarrow$  invar (merge q1 q2)

end
```

46 Heaps

```
theory Heaps
imports
  HOL-Library.Tree_Multiset
  Priority_Queue_Specs
begin
```


Heap = priority queue on trees:

```

locale Heap =
fixes insert :: ('a::linorder)  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree
and del_min :: 'a tree  $\Rightarrow$  'a tree
assumes mset_insert: heap q  $\implies$  mset_tree (insert x q) = {#x#} +
mset_tree q
and mset_del_min:  $\llbracket$  heap q; q  $\neq$  Leaf  $\rrbracket \implies$  mset_tree (del_min q) =
mset_tree q - {#value q#}
and heap_insert: heap q  $\implies$  heap(insert x q)
and heap_del_min: heap q  $\implies$  heap(del_min q)
begin

definition empty :: 'a tree where
empty = Leaf

fun is_empty :: 'a tree  $\Rightarrow$  bool where
is_empty t = (t = Leaf)

fun get_min :: 'a tree  $\Rightarrow$  'a where
get_min (Node l a r) = a

sublocale Priority_Queue where empty = empty and is_empty = is_empty
and insert = insert
and get_min = get_min 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 (auto simp add: mset_del_min neq_Leaf_iff)
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  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree
assumes mset_merge:  $\llbracket \text{heap } q1; \text{heap } q2 \rrbracket \Longrightarrow \text{mset\_tree } (\text{merge } q1 \ q2)$ 
 $= \text{mset\_tree } q1 + \text{mset\_tree } q2$ 
and invar_merge:  $\llbracket \text{heap } q1; \text{heap } q2 \rrbracket \Longrightarrow \text{heap } (\text{merge } q1 \ q2)$ 
begin

fun insert :: 'a  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree where
insert x t = merge (Node Leaf x Leaf) t

fun del_min :: 'a tree  $\Rightarrow$  '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

lemmas local_empty_def = local.empty_def
lemmas local_get_min_def = local.get_min.simps

sublocale PQM: Priority_Queue_Merge where empty = empty and is_empty
= is_empty and insert = insert
and get_min = get_min 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

47 Leftist Heap

theory *Leftist_Heap*

imports

HOL-Library.Pattern_Aliases

Tree2

Priority_Queue_Specs

Complex_Main

HOL-Library.Time_Commands

begin

fun *mset_tree* :: ('a*'b) tree \Rightarrow '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* \Rightarrow nat **where**

mht Leaf = 0

mht (Node _ (_, *n*) _) = *n*

The invariants:

fun (in *linorder*) *heap* :: ('a*'b) tree \Rightarrow bool **where**

heap Leaf = True

heap (Node *l* (*m*, _) *r*) =

$((\forall x \in \text{set_tree } l \cup \text{set_tree } r. m \leq x) \wedge \text{heap } l \wedge \text{heap } r)$

fun *ltree* :: 'a *lheap* \Rightarrow bool **where**

ltree Leaf = True

ltree (Node *l* (*a*, *n*) *r*) =

$(\text{min_height } l \geq \text{min_height } r \wedge n = \text{min_height } r + 1 \wedge \text{ltree } l \ \& \ \text{ltree } r)$

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

empty = Leaf

definition *node* :: 'a *lheap* \Rightarrow 'a \Rightarrow 'a *lheap* \Rightarrow 'a *lheap* **where**

node l a r =

(let *mhl* = *mht l*; *mhr* = *mht r*

in if *mhl* \geq *mhr* then Node *l* (*a*, *mhr*+1) *r* else Node *r* (*a*, *mhl*+1) *l*)

fun *get_min* :: 'a *lheap* \Rightarrow 'a **where**

$get_min(Node\ l\ (a, n)\ r) = a$

For function *merge*:

unbundle *pattern_aliases*

```
fun merge :: 'a::ord heap  $\Rightarrow$  'a heap  $\Rightarrow$  'a heap where
merge Leaf t = t |
merge t Leaf = t |
merge (Node l1 (a1, n1) r1 =: t1) (Node l2 (a2, n2) r2 =: t2) =
  (if a1  $\leq$  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, _)  $\Rightarrow$  t2 |
  (_, Leaf)  $\Rightarrow$  t1 |
  (Node l1 (a1, n1) r1, Node l2 (a2, n2) r2)  $\Rightarrow$ 
    if a1  $\leq$  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  $\Rightarrow$  'a heap  $\Rightarrow$  'a heap where
insert x t = merge (Node Leaf (x,1) Leaf) t
```

```
fun del_min :: 'a::ord heap  $\Rightarrow$  'a heap where
del_min Leaf = Leaf |
del_min (Node l _ r) = merge l r
```

47.1 Lemmas

```
lemma mset_tree_empty: mset_tree t = {#}  $\longleftrightarrow$  t = Leaf
by(cases t) auto
```

```
lemma mht_eq_min_height: ltree t  $\Longrightarrow$  mht t = min_height t
by(cases t) auto
```

```
lemma ltree_node: ltree (node l a r)  $\longleftrightarrow$  ltree l  $\wedge$  ltree r
by(auto simp add: node_def mht_eq_min_height)
```

```
lemma heap_node: heap (node l a r)  $\longleftrightarrow$ 
  heap l  $\wedge$  heap r  $\wedge$  ( $\forall x \in \text{set\_tree } l \cup \text{set\_tree } r. a \leq x$ )
by(auto simp add: node_def)
```

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

47.2 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*: $\llbracket \text{heap } t; t \neq \text{Leaf} \rrbracket \implies \text{get_min } t = \text{Min}(\text{set_tree } t)$
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*: $\llbracket \text{ltree } l; \text{ltree } r \rrbracket \implies \text{ltree } (\text{merge } l r)$
by(*induction l r rule: merge.induct*)(*auto simp: ltree_node*)

lemma *heap_merge*: $\llbracket \text{heap } l; \text{heap } r \rrbracket \implies \text{heap } (\text{merge } l r)$
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 \implies ltree(insert x t)*
by(*simp add: insert_def ltree_merge del: merge.simps split: tree.split*)

lemma *heap_insert*: *heap t \implies heap(insert x t)*
by(*simp add: insert_def heap_merge del: merge.simps split: tree.split*)

lemma *ltree_del_min*: *ltree t \implies ltree(del_min t)*
by(*cases t*)(*auto simp add: ltree_merge simp del: merge.simps*)

lemma *heap_del_min*: *heap t \implies 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 =  $\lambda t. t = \text{Leaf}$ 
and insert = insert and del_min = del_min
and get_min = get_min and merge = merge
and invar =  $\lambda t. \text{heap } t \wedge \text{ltree } t$  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 ltrees_insert)
next
  case 8 thus ?case by(simp add: heap_del_min ltrees_del_min)
next
  case 9 thus ?case by (simp add: mset_merge)
next
  case 10 thus ?case by (simp add: heap_merge ltrees_merge)
qed

```

47.3 Complexity

Auxiliary time functions (which are both 0):

```

time_fun mht
time_fun node

```

```

lemma T_mht_0[simp]: T_mht t = 0
by(cases t)auto

```

Define timing function

```

time_fun merge
time_fun insert
time_fun del_min

```

```

lemma T_merge_min_height: ltree l  $\implies$  ltree r  $\implies$  T_merge l r  $\leq$  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 $T_merge\ l\ r \leq \log 2\ (size1\ l) + \log 2\ (size1\ r) + 1$
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*: $ltree\ t \implies T_insert\ x\ t \leq \log 2\ (size1\ t) + 2$
using *T_merge_log[of Node Leaf (x, 1) Leaf t]*
by(*simp split: tree.split*)

corollary *T_del_min_log*: **assumes** *ltree t*
shows $T_del_min\ t \leq 2 * \log 2\ (size1\ t) + 1$
proof(*cases t rule: tree2_cases*)
case *Leaf* **thus** *?thesis* **using** *assms* **by** *simp*
next
case [*simp*]: (*Node l _ _ r*)
show *?thesis*
using $\langle ltree\ t \rangle\ T_merge_log[of\ l\ r]$
 $log_mono[of\ 2\ size1\ l\ size1\ t]\ log_mono[of\ 2\ size1\ r\ size1\ t]$
by (*simp del: T_merge.simps*)
qed

end

theory *Leftist_Heap_List*
imports
Leftist_Heap
Complex_Main
begin

47.4 Converting a list into a leftist heap

fun *merge_adj* :: (*'a::ord*) *lheap list* \Rightarrow *'a lheap list* **where**
merge_adj [] = [] |
merge_adj [t] = [t] |
merge_adj (t1 # t2 # ts) = *merge* t1 t2 # *merge_adj* ts

For the termination proof of *merge_all* below.

lemma *length_merge_adjacent[termination_simp]*: $length\ (merge_adj\ ts)$
 $= (length\ ts + 1)\ div\ 2$

by (*induction ts rule: merge_adj.induct*) *auto*

fun *merge_all* :: ('a::ord) *lheap list* \Rightarrow 'a *lheap* **where**
merge_all [] = *Leaf* |
merge_all [t] = t |
merge_all ts = *merge_all* (*merge_adj ts*)

47.4.1 Functional correctness

lemma *heap_merge_adj*: $\forall t \in \text{set } ts. \text{heap } t \Longrightarrow \forall t \in \text{set } (\text{merge_adj } ts).$
heap t

by(*induction ts rule: merge_adj.induct*) (*auto simp: heap_merge*)

lemma *ltree_merge_adj*: $\forall t \in \text{set } ts. \text{ltree } t \Longrightarrow \forall t \in \text{set } (\text{merge_adj } ts).$
ltree t

by(*induction ts rule: merge_adj.induct*) (*auto simp: ltree_merge*)

lemma *heap_merge_all*: $\forall t \in \text{set } ts. \text{heap } t \Longrightarrow \text{heap } (\text{merge_all } ts)$

apply(*induction ts rule: merge_all.induct*)

using [[*simp_depth_limit=3*]] **by** (*auto simp add: heap_merge_adj*)

lemma *ltree_merge_all*: $\forall t \in \text{set } ts. \text{ltree } t \Longrightarrow \text{ltree } (\text{merge_all } ts)$

apply(*induction ts rule: merge_all.induct*)

using [[*simp_depth_limit=3*]] **by** (*auto simp add: ltree_merge_adj*)

lemma *mset_merge_adj*:

$$\sum \# (\text{image_mset } \text{mset_tree } (\text{mset } (\text{merge_adj } ts))) = \\ \sum \# (\text{image_mset } \text{mset_tree } (\text{mset } ts))$$

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

lemma *mset_merge_all*:

$$\text{mset_tree } (\text{merge_all } ts) = \sum \# (\text{mset } (\text{map } \text{mset_tree } ts))$$

by(*induction ts rule: merge_all.induct*) (*auto simp: mset_merge mset_merge_adj*)

fun *lheap_list* :: 'a::ord *list* \Rightarrow 'a *lheap* **where**

lheap_list xs = *merge_all* (*map* ($\lambda x. \text{Node Leaf } (x,1) \text{ Leaf}$) *xs*)

lemma *mset_lheap_list*: *mset_tree* (*lheap_list xs*) = *mset xs*

by (*simp add: mset_merge_all o_def*)

lemma *ltree_lheap_list*: *ltree* (*lheap_list ts*)

by(*simp add: ltree_merge_all*)

lemma *heap_lheap_list*: *heap* (*lheap_list ts*)

by(*simp add: heap_merge_all*)

lemma *size_merge*: $\text{size}(\text{merge } t1 \ t2) = \text{size } t1 + \text{size } t2$
by(*induction t1 t2 rule: merge.induct*) (*auto simp: node_def*)

47.4.2 Running time

Not defined automatically because we only count the time for *merge*.

fun *T_merge_adj* :: ('a::ord) *lheap list* \Rightarrow *nat* **where**
T_merge_adj [] = 0 |
T_merge_adj [t] = 0 |
T_merge_adj (t1 # t2 # ts) = *T_merge* t1 t2 + *T_merge_adj* ts

fun *T_merge_all* :: ('a::ord) *lheap list* \Rightarrow *nat* **where**
T_merge_all [] = 0 |
T_merge_all [t] = 0 |
T_merge_all ts = *T_merge_adj* ts + *T_merge_all* (*merge_adj* ts)

fun *T_lheap_list* :: 'a::ord *list* \Rightarrow *nat* **where**
T_lheap_list xs = *T_merge_all* (*map* ($\lambda x.$ *Node Leaf* (x,1) *Leaf*) xs)

abbreviation *Tm* **where**
Tm n == 2 * *log* 2 (n+1) + 1

lemma *T_merge_adj*: $\llbracket \forall t \in \text{set } ts. \text{ltree } t; \forall t \in \text{set } ts. \text{size } t = n \rrbracket$
 $\implies \text{T_merge_adj } ts \leq (\text{length } ts \text{ div } 2) * \text{Tm } n$

proof(*induction ts rule: T_merge_adj.induct*)

case 1 **thus** ?*case* **by** *simp*

next

case 2 **thus** ?*case* **by** *simp*

next

case (3 t1 t2) **thus** ?*case* **using** *T_merge_log*[*of t1 t2*] **by** (*simp add:*
algebra_simps size1_size)

qed

lemma *size_merge_adj*:

$\llbracket \text{even}(\text{length } ts); \forall t \in \text{set } ts. \text{ltree } t; \forall t \in \text{set } ts. \text{size } t = n \rrbracket$
 $\implies \forall t \in \text{set } (\text{merge_adj } ts). \text{size } t = 2*n$

by(*induction ts rule: merge_adj.induct*) (*auto simp: size_merge*)

lemma *T_merge_all*:

$\llbracket \forall t \in \text{set } ts. \text{ltree } t; \forall t \in \text{set } ts. \text{size } t = n; \text{length } ts = 2^k \rrbracket$
 $\implies \text{T_merge_all } ts \leq (\sum_{i=1..k} 2^{k-i} * \text{Tm}(2^{i-1} * n))$

proof (*induction ts arbitrary: k n rule: merge_all.induct*)

```

    case 1 thus ?case by simp
next
    case 2 thus ?case by simp
next
    case (3 t1 t2 ts)
    let ?ts = t1 # t2 # ts
    let ?ts2 = merge_adj ?ts
    obtain k' where k': k = Suc k' using 3.premis(3)
    by (metis length_Cons nat.inject nat_power_eq_Suc_0_iff nat.exhaust)
    have 1:  $\forall x \in \text{set}(\text{merge\_adj } ?ts). \text{ltree } x$ 
    by (rule ltree_merge_adj[OF 3.premis(1)])
    have even (length ts) using 3.premis(3) even_Suc_Suc_iff by fastforce
    from 3.premis(2) size_merge_adj[OF this] 3.premis(1)
    have 2:  $\forall x \in \text{set}(\text{merge\_adj } ?ts). \text{size } x = 2 * n$  by (auto simp: size_merge)
    have 3:  $\text{length } ?ts2 = 2^{\wedge k'}$  using 3.premis(3) k' by (simp add: length_merge_adjacent)
    have 4:  $\text{length } ?ts \text{ div } 2 = 2^{\wedge k'}$ 
    using 3.premis(3) k' by (simp add: power_eq_if[of 2 k] split: if_splits)
    have T_merge_all ?ts = T_merge_adj ?ts + T_merge_all ?ts2 by simp
    also have ...  $\leq 2^{\wedge k'} * Tm \ n + T\_merge\_all \ ?ts2$ 
    using 4 T_merge_adj[OF 3.premis(1,2)] by auto
    also have ...  $\leq 2^{\wedge k'} * Tm \ n + (\sum i=1..k'. 2^{\wedge(k'-i)} * Tm(2^{\wedge(i-1)} * (2*n)))$ 
    using 3.IH[OF 1 2 3] by simp
    also have ...  $= 2^{\wedge k'} * Tm \ n + (\sum i=1..k'. 2^{\wedge(k'-i)} * Tm(2^{\wedge(Suc(i-1))} * n))$ 
    by (simp add: mult_ac cong del: sum.cong)
    also have ...  $= 2^{\wedge k'} * Tm \ n + (\sum i=1..k'. 2^{\wedge(k'-i)} * Tm(2^{\wedge i} * n))$ 
    by (simp)
    also have ...  $= (\sum i=1..k. 2^{\wedge(k-i)} * Tm(2^{\wedge(i-1)} * \text{real } n))$ 
    by (simp add: sum.atLeast_Suc_atMost[of Suc 0 Suc k'] sum.atLeast_Suc_atMost_Suc_shift[of
    _ Suc 0] k'
    del: sum.cl_ivl_Suc)
    finally show ?case .
qed

```

lemma summation: $(\sum i=1..k. 2^{\wedge(k-i)} * ((2::\text{real})*i+1)) = 5*2^{\wedge k} - (2::\text{real})*k - 5$

proof (induction k)

```

    case 0 thus ?case by simp
next

```

```

    case (Suc k)
    have  $(\sum i=1..Suc \ k. 2^{\wedge(Suc \ k - i)} * ((2::\text{real})*i+1))$ 
    =  $(\sum i=1..k. 2^{\wedge(k+1-i)} * ((2::\text{real})*i+1)) + 2*k+3$ 
    by (simp)

```

```

    also have ... = (∑ i=1..k. (2::real)*(2k-i * ((2::real)*i+1))) +
2*k+3
    by (simp add: Suc_diff_le mult.assoc)
    also have ... = 2*(∑ i=1..k. 2k-i * ((2::real)*i+1)) + 2*k+3
    by (simp add: sum_distrib_left)
    also have ... = (2::real)*(5*2k - (2::real)*k - 5) + 2*k+3
    using Suc.IH by simp
    also have ... = 5*2(Suc k) - (2::real)*(Suc k) - 5
    by simp
    finally show ?case .
qed

```

```

lemma T_lheap_list: assumes length xs = 2k
shows T_lheap_list xs ≤ 5 * length xs - 2 * log 2 (length xs)
proof -
  let ?ts = map (λx. Node Leaf (x,1) Leaf) xs
  have T_lheap_list xs = T_merge_all ?ts by simp
  also have ... ≤ (∑ i = 1..k. 2k-i * (2 * log 2 (2i-1 + 1) + 1))
    using T_merge_all[of ?ts 1 k] assms by (simp)
  also have ... ≤ (∑ i = 1..k. 2k-i * (2 * log 2 (2*2i-1) + 1))
    apply (rule sum_mono)
    using zero_le_power[of 2::real] by (simp add: add_pos_nonneg)
  also have ... = (∑ i = 1..k. 2k-i * (2 * log 2 (21+(i-1))) + 1))
    by (simp del: Suc_pred)
  also have ... = (∑ i = 1..k. 2k-i * (2 * log 2 (2i) + 1))
    by (simp)
  also have ... = (∑ i = 1..k. 2k-i * ((2::real)*i+1))
    by (simp add: log_nat_power algebra_simps)
  also have ... = 5*(2::real)k - (2::real)*k - 5
    using summation by (simp)
  finally show ?thesis
    using assms of_nat_le_iff by simp
qed

```

end

48 Binomial Priority Queue

```

theory Binomial_Heap
imports
  HOL-Library.Pattern_Aliases
  Complex_Main
  Priority_Queue_Specs

```

begin

We formalize the 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.

48.1 Binomial Tree and Forest Types

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

type_synonym 'a forest = 'a tree list

48.1.1 Multiset of elements

fun mset_tree :: 'a::linorder tree \Rightarrow 'a multiset **where**
 mset_tree (Node _ a ts) = {#a#} + (\sum t \in #mset ts. mset_tree t)

definition mset_forest :: 'a::linorder forest \Rightarrow 'a multiset **where**
 mset_forest ts = (\sum t \in #mset ts. mset_tree t)

lemma mset_tree_simp_alt[simp]:
 mset_tree (Node r a ts) = {#a#} + mset_forest ts
unfolding mset_forest_def **by** auto
declare mset_tree.simps[simp del]

lemma mset_tree_nonempty[simp]: mset_tree t \neq {#}
by (cases t) auto

lemma mset_forest_Nil[simp]:
 mset_forest [] = {#}
by (auto simp: mset_forest_def)

lemma mset_forest_Cons[simp]: mset_forest (t#ts) = mset_tree t + mset_forest ts
by (auto simp: mset_forest_def)

lemma mset_forest_empty_iff[simp]: mset_forest ts = {#} \longleftrightarrow ts=[]
by (auto simp: mset_forest_def)

lemma root_in_mset[simp]: root t \in # mset_tree t
by (cases t) auto

lemma mset_forest_rev_eq[simp]: mset_forest (rev ts) = mset_forest ts

by (*auto simp: mset_forest_def*)

48.1.2 Invariants

Binomial tree

fun *btree* :: '*a*::*linorder* *tree* \Rightarrow *bool* **where**
btree (*Node* *r* *x* *ts*) \longleftrightarrow
 $(\forall t \in \text{set } ts. \text{btree } t) \wedge \text{map rank } ts = \text{rev } [0..<r]$

Heap invariant

fun *heap* :: '*a*::*linorder* *tree* \Rightarrow *bool* **where**
heap (*Node* $_$ *x* *ts*) $\longleftrightarrow (\forall t \in \text{set } ts. \text{heap } t \wedge x \leq \text{root } t)$

definition *bheap* *t* $\longleftrightarrow \text{btree } t \wedge \text{heap } t$

Binomial Forest invariant:

definition *invar* *ts* $\longleftrightarrow (\forall t \in \text{set } ts. \text{bheap } t) \wedge (\text{sorted_wrt } (<) (\text{map rank } ts))$

A binomial forest is often called a binomial heap, but this overloads the latter term.

The children of a binomial heap node are a valid forest:

lemma *invar_children*:
 $\text{bheap } (\text{Node } r \ v \ ts) \implies \text{invar } (\text{rev } ts)$
by (*auto simp: bheap_def invar_def rev_map[symmetric]*)

48.2 Operations and Their Functional Correctness

48.2.1 *link*

context

includes *pattern_aliases*

begin

fun *link* :: ('*a*::*linorder*) *tree* \Rightarrow '*a* *tree* \Rightarrow '*a* *tree* **where**
 $\text{link } (\text{Node } r \ x_1 \ ts_1 =: t_1) (\text{Node } r' \ x_2 \ ts_2 =: t_2) =$
 $(\text{if } x_1 \leq x_2 \text{ then } \text{Node } (r+1) \ x_1 \ (t_2 \# ts_1) \text{ else } \text{Node } (r+1) \ x_2 \ (t_1 \# ts_2))$

end

lemma *invar_link*:
assumes *bheap* *t*₁
assumes *bheap* *t*₂
assumes *rank* *t*₁ = *rank* *t*₂
shows *bheap* (*link* *t*₁ *t*₂)

using *assms* **unfolding** *bheap_def*
by (*cases* (t_1, t_2) *rule: link.cases*) *auto*

lemma *rank_link[simp]*: $\text{rank } (\text{link } t_1 \ t_2) = \text{rank } t_1 + 1$
by (*cases* (t_1, t_2) *rule: link.cases*) *simp*

lemma *mset_link[simp]*: $\text{mset_tree } (\text{link } t_1 \ t_2) = \text{mset_tree } t_1 + \text{mset_tree } t_2$
by (*cases* (t_1, t_2) *rule: link.cases*) *simp*

48.2.2 *ins_tree*

fun *ins_tree* :: '*a*::*linorder* *tree* \Rightarrow '*a* *forest* \Rightarrow '*a* *forest* **where**
ins_tree *t* [] = [*t*]
| *ins_tree* t_1 ($t_2 \# ts$) =
(if $\text{rank } t_1 < \text{rank } t_2$ then $t_1 \# t_2 \# ts$ else *ins_tree* ($\text{link } t_1 \ t_2$) *ts*)

lemma *bheap0[simp]*: *bheap* (*Node* 0 *x* [])
unfolding *bheap_def* **by** *auto*

lemma *invar_Cons[simp]*:
invar ($t \# ts$)
 $\longleftrightarrow \text{bheap } t \wedge \text{invar } ts \wedge (\forall t' \in \text{set } ts. \text{rank } t < \text{rank } t')$
by (*auto simp: invar_def*)

lemma *invar_ins_tree*:
assumes *bheap* *t*
assumes *invar* *ts*
assumes $\forall t' \in \text{set } ts. \text{rank } t \leq \text{rank } t'$
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_forest_ins_tree[simp]*:
 $\text{mset_forest } (\text{ins_tree } t \ ts) = \text{mset_tree } t + \text{mset_forest } ts$
by (*induction* *t* *ts* *rule: ins_tree.induct*) *auto*

lemma *ins_tree_rank_bound*:
assumes $t' \in \text{set } (\text{ins_tree } t \ ts)$
assumes $\forall t' \in \text{set } ts. \text{rank } t_0 < \text{rank } t'$
assumes $\text{rank } t_0 < \text{rank } t$
shows $\text{rank } t_0 < \text{rank } t'$
using *assms*
by (*induction* *t* *ts* *rule: ins_tree.induct*) (*auto split: if_splits*)

48.2.3 *insert*

hide_const (**open**) *insert*

definition *insert* :: 'a::linorder \Rightarrow 'a forest \Rightarrow 'a forest **where**
insert *x* *ts* = *ins_tree* (*Node* 0 *x* []) *ts*

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

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

48.2.4 *merge*

context

includes *pattern_aliases*

begin

fun *merge* :: 'a::linorder forest \Rightarrow 'a forest \Rightarrow 'a forest **where**
 merge *ts*₁ [] = *ts*₁
| *merge* [] *ts*₂ = *ts*₂
| *merge* (*t*₁*#ts*₁ =: *f*₁) (*t*₂*#ts*₂ =: *f*₂) = (
 if *rank* *t*₁ < *rank* *t*₂ then *t*₁ *# merge* *ts*₁ *f*₂ else
 if *rank* *t*₂ < *rank* *t*₁ then *t*₂ *# merge* *f*₁ *ts*₂
 else *ins_tree* (*link* *t*₁ *t*₂) (*merge* *ts*₁ *ts*₂)
)
end

end

lemma *merge_simp2[simp]*: *merge* [] *ts*₂ = *ts*₂
by (*cases* *ts*₂) *auto*

lemma *merge_rank_bound*:

assumes *t'* \in *set* (*merge* *ts*₁ *ts*₂)

assumes $\forall t_{12} \in \text{set } ts_1 \cup \text{set } ts_2. \text{rank } t < \text{rank } t_{12}$

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 ts2
shows invar (merge ts1 ts2)
using assms
by (induction ts1 ts2 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 ts1
assumes invar ts2
shows invar (merge ts1 ts2)
using assms
proof (induction ts1 ts2 rule: merge.induct)
  case ( $\exists t_1 ts_1 t_2 ts_2$ )
    — Invariants of the parts can be shown automatically
    from 3.prems have [simp]:
      bheap t1 bheap t2

    by auto

```

— These are the three cases of the *merge* function

```

consider (LT) rank t1 < rank t2
    | (GT) rank t1 > rank t2
    | (EQ) rank t1 = rank t2
using antisym_conv3 by blast
then show ?case proof cases
  case LT
    — merge takes the first tree from the left heap
    then have merge (t1 # ts1) (t2 # ts2) = t1 # merge ts1 (t2 # ts2) by
simp
    also have invar ... proof (simp, intro conjI)
      — Invariant follows from induction hypothesis
      show invar (merge ts1 (t2 # ts2))
        using LT 3.IH 3.prems by simp

```

— It remains to show that *t₁* has smallest rank.

```

show  $\forall t' \in \text{set } (\text{merge } ts_1 (t_2 \# ts_2)). \text{rank } t_1 < \text{rank } t'$ 
  — 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.prem3 3.IH by (force elim!: merge_rank_bound)
next
    case [simp]: EQ
    — merge links both first forest, and inserts them into the merged re-
    maining heaps
    have merge (t1 # ts1) (t2 # ts2) = ins_tree (link t1 t2) (merge ts1 ts2)
by simp
    also have invar ... proof (intro invar_ins_tree invar_link)
    — Invariant of merged remaining heaps follows by IH
    show invar (merge ts1 ts2)
    using EQ 3.prem3 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 ∀ t' ∈ set (merge ts1 ts2). rank (link t1 t2) ≤ rank t'
    proof —
    — Which is, again, done with the help of merge_rank_bound
    have rank (link t1 t2) = Suc (rank t2) by simp
    thus ?thesis using 3.prem3 by (auto simp: Suc_le_eq elim!:
merge_rank_bound)
    qed
    qed simp_all
    finally show ?thesis .
    qed
qed auto

```

```

lemma mset_forest_merge[simp]:
  mset_forest (merge ts1 ts2) = mset_forest ts1 + mset_forest ts2
by (induction ts1 ts2 rule: merge.induct) auto

```

48.2.5 get_min

```

fun get_min :: 'a::linorder forest ⇒ 'a where
  get_min [t] = root t
| get_min (t#ts) = min (root t) (get_min ts)

```

```

lemma bheap_root_min:
  assumes bheap t
  assumes x ∈# mset_tree t
  shows root t ≤ x
using assms unfolding bheap_def

```

by (*induction* t *arbitrary*: x *rule*: *mset_tree.induct*) (*fastforce simp*: *mset_forest_def*)

lemma *get_min_mset*:
assumes $ts \neq []$
assumes *invar* ts
assumes $x \in \# \text{mset_forest } ts$
shows $\text{get_min } ts \leq x$
using *assms*
apply (*induction* ts *arbitrary*: x *rule*: *get_min.induct*)
apply (*auto*
simp: *bheap_root_min min_def intro*: *order_trans*;
meson linear order_trans bheap_root_min
 $) +$
done

lemma *get_min_member*:
 $ts \neq [] \implies \text{get_min } ts \in \# \text{mset_forest } ts$
by (*induction* ts *rule*: *get_min.induct*) (*auto simp*: *min_def*)

lemma *get_min*:
assumes $\text{mset_forest } ts \neq \{\#\}$
assumes *invar* ts
shows $\text{get_min } ts = \text{Min_mset } (\text{mset_forest } ts)$
using *assms get_min_member get_min_mset*
by (*auto simp*: *eq_Min_iff*)

48.2.6 *get_min_rest*

fun *get_min_rest* :: $'a::\text{linorder forest} \Rightarrow 'a \text{ tree} \times 'a \text{ forest}$ **where**
 $\text{get_min_rest } [t] = (t, [])$
 $| \text{get_min_rest } (t \# ts) = (\text{let } (t', ts') = \text{get_min_rest } ts$
 $\text{in if root } t \leq \text{root } t' \text{ then } (t, ts) \text{ else } (t', t \# ts'))$

lemma *get_min_rest_get_min_same_root*:
assumes $ts \neq []$
assumes $\text{get_min_rest } ts = (t', ts')$
shows $\text{root } t' = \text{get_min } ts$
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 $\text{get_min_rest } ts = (t', ts')$
assumes $ts \neq []$

```

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  $ts \neq []$ 
  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  $ts \neq []$ 
  assumes invar ts
  shows bheap  $t'$  and invar  $ts'$ 
proof –
  have  $bheap\ t' \wedge invar\ ts'$ 
  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 bheap  $t'$  and invar  $ts'$  by auto
qed

```

48.2.7 *del_min*

definition *del_min* :: '*a*::*linorder* forest \Rightarrow '*a*::*linorder* forest **where**
del_min *ts* = (*case* *get_min_rest* *ts* *of*
 (*Node* *r* *x* *ts*₁, *ts*₂) \Rightarrow *merge* (*itrev* *ts*₁ []) *ts*₂)

```

lemma invar_del_min[simp]:
  assumes  $ts \neq []$ 
  assumes invar ts
  shows invar (del_min ts)
using assms
unfolding del_min_def itrev_Nil
by (auto
  split: prod.split tree.split

```

```

      intro!: invar_merge invar_children
      dest: invar_get_min_rest
    )

lemma mset_forest_del_min:
  assumes  $ts \neq []$ 
  shows  $mset\_forest\ ts = mset\_forest\ (del\_min\ ts) + \{\# get\_min\ ts \#\}$ 
using assms
unfolding del_min_def itrev_Nil
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_forest_def)
done

```

48.2.8 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 bheaps: 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_forest
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_forest_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

```

48.3 Complexity

The size of a binomial tree is determined by its rank

```

lemma size_mset_btree:
  assumes 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_forest 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 bheap t
  shows size (mset_tree t) = 2^rank t
using assms unfolding bheap_def
by (simp add: size_mset_btree)

```

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

```

lemma size_mset_forest:
  assumes invar ts
  shows length ts ≤ log 2 (size (mset_forest ts) + 1)
proof -

```

```

from ⟨invar ts⟩ have
  ASC: sorted_wrt (<) (map rank ts) and
  TINV:  $\forall t \in \text{set } ts. \text{bheap } t$ 
  unfolding invar_def by auto

have  $(2::\text{nat})^{\text{length } ts} = (\sum i \in \{0..<\text{length } ts\}. 2^i) + 1$ 
  by (simp add: sum_power2)
also have  $\dots = (\sum i \leftarrow [0..<\text{length } ts]. 2^i) + 1$  (is  $\_ = ?S + 1$ )
  by (simp add: interv_sum_list_conv_sum_set_nat)
also have  $?S \leq (\sum t \leftarrow ts. 2^{\text{rank } t})$  (is  $\_ \leq ?T$ )
  using sorted_wrt_less_idx[OF ASC] by(simp add: sum_list_mono2)
also have  $?T + 1 \leq (\sum t \leftarrow ts. \text{size } (\text{mset\_tree } t)) + 1$  using TINV
  by (auto cong: map_cong simp: size_mset_tree)
also have  $\dots = \text{size } (\text{mset\_forest } ts) + 1$ 
  unfolding mset_forest_def by (induction ts) auto
finally have  $2^{\text{length } ts} \leq \text{size } (\text{mset\_forest } ts) + 1$  by simp
then show ?thesis using le_log2_of_power by blast
qed

```

48.3.1 Timing Functions

time_fun *link*

lemma *T_link*[*simp*]: $T_link\ t_1\ t_2 = 0$
by(*cases t₁*; *cases t₂*, *auto*)

time_fun *rank*

lemma *T_rank*[*simp*]: $T_rank\ t = 0$
by(*cases t*, *auto*)

time_fun *ins_tree*

time_fun *insert*

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

48.3.2 *T_insert*

lemma *T_insert_bound*:

assumes *invar ts*

shows $T_insert\ x\ ts \leq \log 2\ (\text{size } (\text{mset_forest } ts) + 1) + 1$

proof –

```

have real (T_insert x ts)  $\leq$  real (length ts) + 1
  unfolding T_insert.simps using T_ins_tree_simple_bound
  by (metis of_nat_1 of_nat_add of_nat_mono)
also note size_mset_forest[OF  $\langle$ invar ts $\rangle$ ]
finally show ?thesis by simp
qed

```

48.3.3 *T_merge*

time_fun *merge*

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*:

```

T_merge ts1 ts2 + length (merge ts1 ts2)  $\leq$  2 * (length ts1 + length ts2)
+ 1
by (induction ts1 ts2 rule: 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  $\equiv$  size (mset_forest ts1)
defines n2  $\equiv$  size (mset_forest ts2)
assumes invar ts1 invar ts2
shows T_merge ts1 ts2  $\leq$  4 * log 2 (n1 + n2 + 1) + 1
proof –
  note n_defs = assms(1,2)

```

```

have T_merge ts1 ts2  $\leq$  2 * real (length ts1) + 2 * real (length ts2) + 1
  using T_merge_length[of ts1 ts2] by simp
also note size_mset_forest[OF  $\langle$ invar ts1 $\rangle$ ]
also note size_mset_forest[OF  $\langle$ invar ts2 $\rangle$ ]
finally have T_merge ts1 ts2  $\leq$  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)  $\leq$  log 2 (n1 + n2 + 1)
  unfolding n_defs by (simp add: algebra_simps)
also have log 2 (n2 + 1)  $\leq$  log 2 (n1 + n2 + 1)
  unfolding n_defs by (simp add: algebra_simps)

```

finally show *?thesis* by (simp add: algebra_simps)
qed

48.3.4 T_get_min

time_fun root

lemma $T_root[simp]$: $T_root\ t = 0$
by(cases t)(simp_all)

time_fun min

time_fun get_min

lemma $T_get_min_estimate$: $ts \neq [] \implies T_get_min\ ts = length\ ts$
by (induction ts rule: $T_get_min.induct$) auto

lemma $T_get_min_bound$:
assumes $invar\ ts$
assumes $ts \neq []$
shows $T_get_min\ ts \leq \log 2\ (size\ (mset_forest\ ts) + 1)$
proof -
have 1: $T_get_min\ ts = length\ ts$ using assms $T_get_min_estimate$ by
auto
also note $size_mset_forest[OF\ \langle invar\ ts \rangle]$
finally show *?thesis* .
qed

48.3.5 T_del_min

time_fun get_min_rest

lemma $T_get_min_rest_estimate$: $ts \neq [] \implies T_get_min_rest\ ts = length\ ts$
by (induction ts rule: $T_get_min_rest.induct$) auto

lemma $T_get_min_rest_bound$:
assumes $invar\ ts$
assumes $ts \neq []$
shows $T_get_min_rest\ ts \leq \log 2\ (size\ (mset_forest\ ts) + 1)$
proof -
have 1: $T_get_min_rest\ ts = length\ ts$ using assms $T_get_min_rest_estimate$
by auto
also note $size_mset_forest[OF\ \langle invar\ ts \rangle]$


```

    finally show ?thesis .
qed

time_fun del_min

lemma T_del_min_bound:
  fixes ts
  defines  $n \equiv \text{size } (\text{mset\_forest } ts)$ 
  assumes  $\text{invar } ts$  and  $ts \neq []$ 
  shows  $T\_del\_min \ ts \leq 6 * \log 2 \ (n+1) + 2$ 
proof -
  obtain  $r \ x \ ts_1 \ ts_2$  where  $GM: \text{get\_min\_rest } ts = (\text{Node } r \ x \ ts_1, \ ts_2)$ 
    by (metis surj_pair tree.exhaust_sel)

  have  $I1: \text{invar } (\text{rev } ts_1)$  and  $I2: \text{invar } ts_2$ 
    using  $\text{invar\_get\_min\_rest}[OF \ GM \ \langle ts \neq [] \rangle \ \langle \text{invar } ts \rangle]$   $\text{invar\_children}$ 
    by auto

  define  $n_1$  where  $n_1 = \text{size } (\text{mset\_forest } ts_1)$ 
  define  $n_2$  where  $n_2 = \text{size } (\text{mset\_forest } ts_2)$ 

  have  $n_1 \leq n$   $n_1 + n_2 \leq n$  unfolding  $n\_def \ n_1\_def \ n_2\_def$ 
    using  $\text{mset\_get\_min\_rest}[OF \ GM \ \langle ts \neq [] \rangle]$ 
    by (auto simp:  $\text{mset\_forest\_def}$ )

  have  $T\_del\_min \ ts = \text{real } (T\_get\_min\_rest \ ts) + \text{real } (T\_itrev \ ts_1 \ [])$ 
    +  $\text{real } (T\_merge \ (\text{rev } ts_1) \ ts_2)$ 
    unfolding  $T\_del\_min.simps \ GM \ T\_itrev \ itrev\_Nil$ 
    by simp
  also have  $T\_get\_min\_rest \ ts \leq \log 2 \ (n+1)$ 
    using  $T\_get\_min\_rest\_bound[OF \ \langle \text{invar } ts \rangle \ \langle ts \neq [] \rangle]$  unfolding  $n\_def$ 
  by simp
  also have  $T\_itrev \ ts_1 \ [] \leq 1 + \log 2 \ (n_1 + 1)$ 
    unfolding  $T\_itrev \ n_1\_def$  using  $\text{size\_mset\_forest}[OF \ I1]$  by simp
  also have  $T\_merge \ (\text{rev } ts_1) \ ts_2 \leq 4 * \log 2 \ (n_1 + n_2 + 1) + 1$ 
    unfolding  $n_1\_def \ n_2\_def$  using  $T\_merge\_bound[OF \ I1 \ I2]$  by (simp
  add:  $\text{algebra\_simps}$ )
  finally have  $T\_del\_min \ ts \leq \log 2 \ (n+1) + \log 2 \ (n_1 + 1) + 4 * \log 2$ 
     $(\text{real } (n_1 + n_2) + 1) + 2$ 
    by (simp add:  $\text{algebra\_simps}$ )
  also note  $\langle n_1 + n_2 \leq n \rangle$ 
  also note  $\langle n_1 \leq n \rangle$ 
  finally show ?thesis by (simp add:  $\text{algebra\_simps}$ )
qed

```

end

49 The Median-of-Medians Selection Algorithm

theory *Selection*

imports *Complex_Main HOL-Library.Time_Functions Sorting*

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*: $\text{replicate } (\text{numeral } n) \ x = x \# \text{replicate } (\text{pred_numeral } n) \ x$

by (*simp add: numeral_eq_Suc*)

lemma *insert_correct*: $\text{insert } xs = \text{sort } xs$

using *sorted_insert mset_insert* **by** (*metis properties_for_sort*)

lemma *sum_list_replicate* [*simp*]: $\text{sum_list } (\text{replicate } n \ x) = n * x$

by (*induction n*) *auto*

lemma *mset_concat*: $\text{mset } (\text{concat } xss) = \text{sum_list } (\text{map } \text{mset } xss)$

by (*induction xss*) *simp_all*

lemma *set_mset_sum_list* [*simp*]: $\text{set_mset } (\text{sum_list } xs) = (\bigcup_{x \in \text{set } xs} \text{set_mset } x)$

by (*induction xs*) *auto*

lemma *filter_mset_image_mset*:

$\text{filter_mset } P \ (\text{image_mset } f \ A) = \text{image_mset } f \ (\text{filter_mset } (\lambda x. P \ (f \ x)) \ A)$

by (*induction A*) *auto*

lemma *filter_mset_sum_list*: $\text{filter_mset } P \ (\text{sum_list } xs) = \text{sum_list } (\text{map } (\text{filter_mset } P) \ xs)$

by (*induction xs*) *simp_all*

lemma *sum_mset_mset_mono*:

assumes $(\bigwedge x. x \in \# A \implies f \ x \subseteq \# g \ x)$

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  $A \subseteq\# B \wedge x. x \in\# A \implies P\ x \implies Q\ x$ 
  shows  $\text{filter\_mset } P\ A \subseteq\# \text{filter\_mset } Q\ B$ 
  by (rule mset_subset_eqI) (insert assms, auto simp: mset_subset_eq_count
count_eq_zero_iff)

lemma size_mset_sum_mset_distrib:  $\text{size } (\text{sum\_mset } A :: 'a \text{ multiset}) =$ 
 $\text{sum\_mset } (\text{image\_mset } \text{size } A)$ 
  by (induction A) auto

lemma sum_mset_mono:
  assumes  $\wedge x. x \in\# A \implies f\ x \leq (g\ x :: 'a :: \{\text{ordered\_ab\_semigroup\_add, comm\_monoid\_add}\})$ 
  shows  $(\sum x \in\# A. f\ x) \leq (\sum x \in\# A. g\ x)$ 
  using assms by (induction A) (auto intro!: add_mono)

lemma filter_mset_is_empty_iff:  $\text{filter\_mset } P\ A = \{\#\} \longleftrightarrow (\forall x. x \in\#$ 
 $A \longrightarrow \neg P\ x)$ 
  by (auto simp: multiset_eq_iff count_eq_zero_iff)

lemma sort_eq_Nil_iff [simp]:  $\text{sort } xs = [] \longleftrightarrow xs = []$ 
  by (metis set_empty set_sort)

lemma sort_mset_cong:  $\text{mset } xs = \text{mset } ys \implies \text{sort } xs = \text{sort } ys$ 
  by (metis sorted_list_of_multiset_mset)

lemma Min_set_sorted:  $\text{sorted } xs \implies xs \neq [] \implies \text{Min } (\text{set } xs) = \text{hd } xs$ 
  by (cases xs; force intro: Min_insert2)

lemma hd_sort:
  fixes xs :: 'a :: linorder list
  shows  $xs \neq [] \implies \text{hd } (\text{sort } xs) = \text{Min } (\text{set } xs)$ 
  by (subst Min_set_sorted [symmetric]) auto

lemma length_filter_conv_size_filter_mset:  $\text{length } (\text{filter } P\ xs) = \text{size } (\text{filter\_mset}$ 
 $P\ (\text{mset } xs))$ 
  by (induction xs) auto

lemma sorted_filter_less_subset_take:
  assumes sorted xs and  $i < \text{length } xs$ 
  shows  $\{\#x \in\# \text{mset } xs. x < xs\ i\#\} \subseteq\# \text{mset } (\text{take } i\ xs)$ 
  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.prem by (auto simp: filter_mset_is_empty_iff)
next
  case (Suc i')
  have {#y ∈# mset (x # xs). y < (x # xs) ! i#} ⊆# add_mset x {#y
    ∈# mset xs. y < xs ! i'#}
    using Suc Cons.prem by (auto)
  also have ... ⊆# add_mset x (mset (take i' xs))
    unfolding mset_subset_eq_add_mset_cancel using Cons.prem Suc
    by (intro Cons.IH) (auto)
  also have ... = mset (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 ∈# mset xs. x > xs ! i#} ⊆# mset (drop (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 ∈# mset (x # xs). y > (x # xs) ! i#} ⊆# {#y ∈# mset xs.
      y > xs ! i'#}
      using Suc Cons.prem by (auto simp: set_conv_nth)
    also have ... ⊆# mset (drop (Suc i') xs)
      using Cons.prem Suc by (intro Cons.IH) (auto)
    also have ... = mset (drop (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
lemmas [simp] = chop.simps(1)
```

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

```
lemma chop_induct' [case_names trivial reduce]:
  assumes  $\bigwedge n \ xs. \ n = 0 \vee xs = [] \implies P \ n \ xs$ 
  assumes  $\bigwedge n \ xs. \ n > 0 \implies xs \neq [] \implies P \ n \ (drop \ n \ xs) \implies P \ n \ xs$ 
  shows  $P \ n \ xs$ 
  using assms
proof induction_schema
  show wf (measure (length  $\circ$  snd))
  by auto
qed (blast | simp)+
```

```
lemma chop_eq_Nil_iff [simp]: chop n xs = []  $\longleftrightarrow$   $n = 0 \vee xs = []$ 
  by (induction n xs rule: chop.induct; subst chop.simps) auto
```

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

```
lemma chop_reduce:  $n > 0 \implies xs \neq [] \implies chop \ n \ xs = take \ n \ xs \# chop \ n \ (drop \ n \ xs)$ 
  by (cases n; cases xs) (auto simp: chop.simps)
```

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

```
lemma chop_elem_not_Nil [dest]:  $ys \in set \ (chop \ n \ xs) \implies ys \neq []$ 
  by (induction n xs rule: chop.induct; subst (asm) chop.simps)
  (auto simp: eq_commute[of []] split: if_splits)
```

```
lemma length_chop_part_le:  $ys \in set \ (chop \ n \ xs) \implies length \ ys \leq n$ 
  by (induction n xs rule: chop.induct; subst (asm) chop.simps) (auto split:
if_splits)
```

```
lemma length_chop:
  assumes  $n > 0$ 
  shows  $length \ (chop \ n \ xs) = nat \ \lceil length \ xs / n \rceil$ 
proof -
  from  $\langle n > 0 \rangle$  have  $real \ n * length \ (chop \ n \ xs) \geq length \ xs$ 
  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 ∀ xs∈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.premys by (intro chop_append) auto
also have chop d xs = [xs]
  using Cons.premys by (subst chop_reduce) auto
also have chop d (concat xss) = xss
  using Cons.premys 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 $select_0: xs \neq [] \implies select\ 0\ xs = Min\ (set\ xs)$
by (simp add: hd_sort select_def flip: hd_conv_nth)

lemma $select_mset_cong: mset\ xs = mset\ ys \implies select\ k\ xs = select\ k\ ys$
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 $size\ \{\#y \in \# mset\ (sort\ xs). y < select\ n\ xs\# \} \leq size\ (mset\ (take\ n\ (sort\ xs)))$

unfolding select_def **using** assms

by (intro size_mset_mono sorted_filter_less_subset_take) auto

thus $size\ \{\#y \in \# mset\ xs. y < select\ n\ xs\# \} \leq n$

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 $\dots = \text{length } xs \text{ div } 2 + 1$
using $\langle \text{length } xs > 0 \rangle$ **by** *linarith*
finally show $\text{size } \{\#y \in \# \text{ mset } xs. y > \text{median } xs\# \} \leq \text{length } xs \text{ div } 2$
by *simp*
qed *auto*

lemmas *median_props* = *size_less_than_median size_greater_than_median*

49.5 A recurrence for selection

definition *partition3* :: $'a \Rightarrow 'a :: \text{linorder list} \Rightarrow 'a \text{ list} \times 'a \text{ list} \times 'a \text{ list}$
where

$\text{partition3 } x \text{ } xs = (\text{filter } (\lambda y. y < x) \text{ } xs, \text{filter } (\lambda y. y = x) \text{ } xs, \text{filter } (\lambda y. y > x) \text{ } xs)$

lemma *partition3_code* [code]:

$\text{partition3 } x \text{ } [] = ([], [], [])$
 $\text{partition3 } x \text{ } (y \# ys) =$
 $(\text{case } \text{partition3 } x \text{ } ys \text{ of } (ls, es, gs) \Rightarrow$
 $\text{if } y < x \text{ then } (y \# ls, es, gs) \text{ else if } x = y \text{ then } (ls, y \# es, gs) \text{ else}$
 $(ls, es, y \# gs))$
by (*auto simp: partition3_def*)

lemma *length_partition3*:

assumes $\text{partition3 } x \text{ } xs = (ls, es, gs)$
shows $\text{length } xs = \text{length } ls + \text{length } es + \text{length } gs$
using *assms* **by** (*induction xs arbitrary: ls es gs*)
 $(\text{auto simp: partition3_code split: if_splits prod.splits})$

lemma *sort_append*:

assumes $\forall x \in \text{set } xs. \forall y \in \text{set } ys. x \leq y$
shows $\text{sort } (xs @ ys) = \text{sort } xs @ \text{sort } ys$
using *assms* **by** (*intro properties_for_sort*) (*auto simp: sorted_append*)

lemma *select_append*:

assumes $\forall y \in \text{set } ys. \forall z \in \text{set } zs. y \leq z$
shows $k < \text{length } ys \implies \text{select } k \text{ } (ys @ zs) = \text{select } k \text{ } ys$
and $k \in \{\text{length } ys.. \text{length } ys + \text{length } zs\} \implies$
 $\text{select } k \text{ } (ys @ zs) = \text{select } (k - \text{length } ys) \text{ } zs$
using *assms* **by** (*simp_all add: select_def sort_append nth_append*)

lemma *select_append'*:

assumes $\forall y \in \text{set } ys. \forall z \in \text{set } zs. y \leq z$ **and** $k < \text{length } ys + \text{length } zs$
shows $\text{select } k \text{ } (ys @ zs) = (\text{if } k < \text{length } ys \text{ then } \text{select } k \text{ } ys \text{ else } \text{select}$

$(k - \text{length } ys) \text{ } zs)$
using *assms* **by** (*auto intro!*: *select_append*)

theorem *select_rec_partition*:

assumes $k < \text{length } xs$
shows $\text{select } k \text{ } xs =$ (
 $\text{let } (ls, es, gs) = \text{partition3 } x \text{ } xs$
in
 $\text{if } k < \text{length } ls \text{ then } \text{select } k \text{ } ls$
 $\text{else if } k < \text{length } ls + \text{length } es \text{ then } x$
 $\text{else } \text{select } (k - \text{length } ls - \text{length } es) \text{ } gs$
 $)$ (**is** $_ = ?rhs$)

proof –

define $ls \text{ } es \text{ } gs$ **where** $ls = \text{filter } (\lambda y. y < x) \text{ } xs$ **and** $es = \text{filter } (\lambda y. y =$
 $x) \text{ } xs$

and $gs = \text{filter } (\lambda y. y > x) \text{ } xs$

define $nl \text{ } ne$ **where** [*simp*]: $nl = \text{length } ls$ $ne = \text{length } es$

have *mset_eq*: $\text{mset } xs = \text{mset } ls + \text{mset } es + \text{mset } gs$

unfolding $ls_def \text{ } es_def \text{ } gs_def$ **by** (*induction xs*) *auto*

have *length_eq*: $\text{length } xs = \text{length } ls + \text{length } es + \text{length } gs$

unfolding $ls_def \text{ } es_def \text{ } gs_def$

using [*simp_depth_limit* = 1]] **by** (*induction xs*) *auto*

have [*simp*]: $\text{select } i \text{ } es = x$ **if** $i < \text{length } es$ **for** i

proof –

have $\text{select } i \text{ } es \in \text{set } (\text{sort } es)$ **unfolding** *select_def*

using *that* **by** (*intro nth_mem*) *auto*

thus *?thesis*

by (*auto simp: es_def*)

qed

have $\text{select } k \text{ } xs = \text{select } k \text{ } (ls @ (es @ gs))$

by (*intro select_mset_cong*) (*simp_all add: mset_eq*)

also have $\dots = (\text{if } k < nl \text{ then } \text{select } k \text{ } ls \text{ else } \text{select } (k - nl) \text{ } (es @ gs))$

unfolding nl_ne_def **using** *assms*

by (*intro select_append'*) (*auto simp: ls_def es_def gs_def length_eq*)

also have $\dots = (\text{if } k < nl \text{ then } \text{select } k \text{ } ls \text{ else if } k < nl + ne \text{ then } x$
 $\text{else } \text{select } (k - nl - ne) \text{ } gs)$

proof (*rule if_cong*)

assume $\neg k < nl$

have $\text{select } (k - nl) \text{ } (es @ gs) =$

$(\text{if } k - nl < ne \text{ then } \text{select } (k - nl) \text{ } es \text{ else } \text{select } (k - nl -$

$ne) \text{ } gs)$

unfolding nl_ne_def **using** *assms* $\langle \neg k < nl \rangle$

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

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 $\{\#y \in \# \text{ mset } xs. y \prec M\# \} = \{\#y \in \# (\sum ys \leftarrow \text{chop } 5 \text{ xs}. \text{mset } ys). y \prec M\# \}$
by $(\text{subst sum_msets_chop}) \text{ simp_all}$
also have $\dots = (\sum ys \leftarrow \text{chop } 5 \text{ xs}. \{\#y \in \# \text{ mset } ys. y \prec M\# \})$
by $(\text{subst filter_mset_sum_list}) (\text{simp add: } o_def)$
also have $\dots = (\sum ys \in \# \text{mset } (\text{chop } 5 \text{ xs}). \{\#y \in \# \text{ mset } ys. y \prec M\# \})$
by $(\text{subst sum_mset_sum_list } [\text{symmetric}]) \text{ simp_all}$
also have $\text{mset } (\text{chop } 5 \text{ xs}) = Y_{\prec} + Y_{\succeq}$
by $(\text{simp add: } Y_small_def Y_big_def \text{not_le})$
also have $(\sum ys \in \# \dots \{\#y \in \# \text{ mset } ys. y \prec M\# \}) =$
 $(\sum ys \in \# Y_{\prec}. \{\#y \in \# \text{ mset } ys. y \prec M\# \}) + (\sum ys \in \# Y_{\succeq}. \{\#y \in \# \text{ mset } ys. y \prec M\# \})$
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 $\dots \subseteq \# (\sum ys \in \# Y_{\prec}. \text{mset } ys) + (\sum ys \in \# Y_{\succeq}. \{\#y \in \# \text{ mset } ys. y \prec \text{median } ys\# \})$
using R
by $(\text{intro subset_mset.add_mono sum_mset_mset_mono mset_filter_mono})$
 $(\text{auto simp: } Y_big_def)$
finally have $\text{size } \{\# y \in \# \text{ mset } xs. y \prec M\# \} \leq \text{size } \dots$
by $(\text{rule size_mset_mono})$
hence $\text{size } \{\# y \in \# \text{ mset } xs. y \prec M\# \} \leq$
 $(\sum ys \in \# Y_{\prec}. \text{length } ys) + (\sum ys \in \# Y_{\succeq}. \text{size } \{\#y \in \# \text{ mset } ys. y \prec \text{median } ys\# \})$
by $(\text{simp add: size_mset_sum_mset_distrib multiset.map_comp } o_def)$

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

also have $(\sum_{ys \in \# Y_{\prec}} \text{length } ys) \leq (\sum_{ys \in \# Y_{\prec}} 5)$
by $(\text{intro sum_mset_mono}) (\text{auto simp: } Y_small_def \text{length_chop_part_le})$
also have $\dots = 5 * \text{size } Y_{\prec}$ **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 $(\sum_{ys \in \# Y_{\succeq}} \text{size } \{\#y \in \# \text{mset } ys. y \prec \text{median } ys\# \}) \leq$
 $(\sum_{ys \in \# Y_{\succeq}} \text{length } ys \text{ div } 2)$

proof $(\text{intro sum_mset_mono, goal_cases})$

fix ys **assume** $ys \in \# Y_{\succeq}$

hence $ys \neq []$

by $(\text{auto simp: } Y_big_def)$

thus $\text{size } \{\#y \in \# \text{mset } ys. y \prec \text{median } ys\# \} \leq \text{length } ys \text{ div } 2$

using $R \text{ median_props[of } ys]$ **by** auto

qed

also have $\dots \leq (\sum_{ys \in \# Y_{\succeq}} 2)$

by $(\text{intro sum_mset_mono div_le_mono diff_le_mono})$

$(\text{auto simp: } Y_big_def \text{dest: length_chop_part_le})$

also have $\dots = 2 * \text{size } Y_{\succeq}$ **by** simp

Simplifying gives us the main result.

also have $5 * \text{size } Y_{\prec} + 2 * \text{size } Y_{\succeq} = 2 * m + 3 * \text{size } Y_{\prec}$

by $(\text{simp add: } m_eq)$

also have $\dots \leq 3.5 * m$

using $\langle \text{size } Y_{\prec} \leq m \text{ div } 2 \rangle$ **by** linarith

also have $\dots = 3.5 * \lceil n / 5 \rceil$

by $(\text{simp add: } m_def \text{length_chop})$

also have $\dots \leq 0.7 * n + 3.5$

by linarith

finally have $\text{size } \{\#y \in \# \text{mset } xs. y \prec M\# \} \leq 0.7 * n + 3.5$

by simp

thus $\text{size } \{\#y \in \# \text{mset } xs. y \prec M\# \} \leq \text{nat } \lceil 0.7 * n + 3 \rceil$

by linarith

qed

lemma $\text{size_less_than_median_of_medians:}$

$\text{size } \{\#y \in \# \text{mset } xs. y < M\# \} \leq \text{nat } \lceil 0.7 * \text{length } xs + 3 \rceil$

using $\text{size_median_of_medians_aux[of } (<)]$ **by** simp

lemma $\text{size_greater_than_median_of_medians:}$

$\text{size } \{\#y \in \# \text{mset } xs. y > M\# \} \leq \text{nat } \lceil 0.7 * \text{length } xs + 3 \rceil$

using $\text{size_median_of_medians_aux[of } (>)]$ **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* = *insort 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 insort_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 = (
 let *n* = *length xs*
 in if *n* ≤ 20 *then*
 slow_select k xs
 else
 let *M* = *mom_select (((n + 4) div 5 - 1) div 2) (map slow_median (chop 5 xs))*;
 (*ls*, *es*, *gs*) = *partition3 M xs*;
 nl = *length ls*
 in
 if *k* < *nl* *then* *mom_select k ls*
 else let *ne* = *length es* *in if* *k* < *nl* + *ne* *then* *M*
 else mom_select (k - nl - ne) 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.prem 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)[OF refl False x_def tw refl refl refl]
            1.IH(3)[OF refl False x_def tw refl refl 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.prem by (simp add: ls_def es_def gs_def nl_def
ne_def)
            try0
            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 $\neg \text{length } xs \leq 20$

thus (((length *xs* + 4) div 5 - 1) div 2, map *slow_median* (chop 5 *xs*)),
k, *xs*)

$\in \text{measure } (\text{length} \circ \text{snd})$

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* = *mom_select* (((length *xs* + 4) div 5 - 1) div 2)
 (map *slow_median* (chop 5 *xs*))

assume *A*: $\neg \text{length } xs \leq 20$

 (*ls*, *es*, *gs*) = *partition3* *x* *xs*

mom_select_dom (((length *xs* + 4) div 5 - 1) div 2,
 map *slow_median* (chop 5 *xs*))

have *less*: ((length *xs* + 4) div 5 - 1) div 2 < nat $\lceil \text{length } xs / 5 \rceil$

using *A*(1) **by** *linarith*

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

have *x* = *select* (((length *xs* + 4) div 5 - 1) div 2) (map *slow_median*
 (chop 5 *xs*))

using *less* **unfolding** *x_def* **by** (*intro mom_select_correct_aux A*)

 (*auto simp: length_chop*)

also have ... $\in \text{set } (\text{map } \text{slow_median } (\text{chop } 5 \text{ } xs))$

using *less* **by** (*intro select_in_set*) (*simp_all add: length_chop*)

also have ... $\subseteq \text{set } xs$

unfolding *set_map*

proof *safe*

fix *ys* **assume** *ys*: *ys* $\in \text{set } (\text{chop } 5 \text{ } xs)$

hence *median ys* $\in \text{set } ys$

by *auto*

also have *set ys* $\subseteq \bigcup (\text{set } ' \text{set } (\text{chop } 5 \text{ } xs))$

using *ys* **by** *blast*

also have ... = *set xs*

by (*rule UN_sets_chop*) *simp_all*

finally show *slow_median ys* $\in \text{set } xs$

by (*simp add: slow_median_correct*)

qed

finally have *x* $\in \text{set } xs$.

thus ((*k*, *ls*), *k*, *xs*) $\in \text{measure } (\text{length} \circ \text{snd})$

and ((*k* - length *ls* - length *es*, *gs*), *k*, *xs*) $\in \text{measure } (\text{length} \circ \text{snd})$

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*: $k < \text{length } xs \implies \text{mom_select } k \text{ } xs = \text{select } k \text{ } xs$
using *mom_select_correct_aux* **and** *mom_select_termination* **by** *blast*

49.8 Running time analysis

time_fun *partition3* **equations** *partition3_code*

lemma *T_partition3*: $T_partition3 \text{ } x \text{ } xs = \text{length } xs + 1$
by (induction $x \text{ } xs$ rule: *T_partition3.induct*) *auto*

time_definition *slow_select*

lemmas *T_slow_select_def* [*simp del*] = *T_slow_select.simps*

time_fun *slow_median*

lemma *T_slow_select_le*:
assumes $k < \text{length } xs$
shows $T_slow_select \text{ } k \text{ } xs \leq \text{length } xs^2 + 3 * \text{length } xs + 1$
proof –
have $T_slow_select \text{ } k \text{ } xs = T_insert \text{ } xs + T_nth \text{ } (Sorting.insert \text{ } xs) \text{ } k$
unfolding *T_slow_select_def* ..
also have $T_insert \text{ } xs \leq (\text{length } xs + 1)^2$
by (rule *T_insert_length*)
also have $T_nth \text{ } (Sorting.insert \text{ } xs) \text{ } k = k + 1$
using *assms* **by** (subst *T_nth*) (auto simp: *length_insert*)
also have $k + 1 \leq \text{length } xs$
using *assms* **by** *linarith*
also have $(\text{length } xs + 1)^2 + \text{length } xs = \text{length } xs^2 + 3 * \text{length } xs + 1$
by (simp add: *algebra_simps* *power2_eq_square*)
finally show ?thesis **by** – *simp_all*
qed

lemma *T_slow_median_le*:

```

assumes  $xs \neq []$ 
shows  $T\_slow\_median\ xs \leq length\ xs^2 + 4 * length\ xs + 2$ 
proof -
  have  $T\_slow\_median\ xs = length\ xs + T\_slow\_select\ ((length\ xs - 1) \div 2)\ xs + 1$ 
    by (simp add: T_length)
  also from assms have  $length\ xs > 0$ 
    by simp
  hence  $(length\ xs - 1) \div 2 < length\ xs$ 
    by linarith
  hence  $T\_slow\_select\ ((length\ xs - 1) \div 2)\ xs \leq length\ xs^2 + 3 * length\ xs + 1$ 
    by (intro T_slow_select_le) auto
  also have  $length\ xs + \dots + 1 = length\ xs^2 + 4 * length\ xs + 2$ 
    by (simp add: algebra_simps)
  finally show ?thesis by - simp_all
qed

```

time_fun *chop*

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*:
 $n > 0 \implies xs \neq [] \implies T_chop\ n\ xs = T_take\ n\ xs + T_drop\ n\ xs + T_chop\ n\ (drop\ n\ xs) + 1$
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 T_drop*)

time_fun *mom_select*

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

lemma *T_mom_select_simps*:
 $length\ xs \leq 20 \implies T_mom_select\ k\ xs = T_slow_select\ k\ xs + T_length$

```

xs + 1
length xs > 20 ==> T_mom_select k xs = (
  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 T_length es + (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 xs + 1
)
by (subst T_mom_select.simps; simp add: Let_def case_prod_unfold)+

```

```

function T'_mom_select :: nat => nat where
  T'_mom_select n =
    (if n ≤ 20 then
      483
    else
      T'_mom_select (nat ⌈0.2*n⌉) + T'_mom_select (nat ⌈0.7*n+3⌉)
+ 19 * n + 54)
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 ≥ 483
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 = 483

```

```

    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⌉) + 19 * m + 54
  by (subst T'_mom_select.simps) auto
  also have ... ≤ T'_mom_select (nat ⌈0.2*n⌉) + T'_mom_select (nat
⌈0.7*n + 3⌉) + 19 * n + 54
    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*:

```

  assumes k < length xs
  shows T_mom_select k xs ≤ T'_mom_select (length xs)
  using assms
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 (((n + 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 ..
  note IH = 1.IH(1)[OF n_def]
    1.IH(2)[OF n_def _ x_def tw refl refl nl_def]
    1.IH(3)[OF n_def _ x_def tw refl refl nl_def _ ne_def]

  show ?case
proof (cases length xs ≤ 20)
  case True — base case
  hence T_mom_select k xs ≤ (length xs)2 + 4 * length xs + 3

```

```

    using T_slow_select_le[of k xs] ⟨k < length xs⟩
    by (subst T_mom_select_simps(1)) (auto simp: T_length)
  also have ... ≤ 202 + 4 * 20 + 3
    using True by (intro add_mono power_mono) auto
  also have ... = 483
    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 ≤ 10 * n + 48
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)
  also have (∑ ys←chop 5 xs. T_slow_median ys) ≤ (∑ ys←chop 5
xs. 47)
    proof (intro sum_list_mono)
      fix ys assume ys ∈ set (chop 5 xs)
      hence length ys ≤ 5 ys ≠ []
        using length_chop_part_le[of ys 5 xs] by auto
      from ⟨ys ≠ []⟩ have T_slow_median ys ≤ (length ys) ^ 2 + 4 *
length ys + 2
        by (rule T_slow_median_le)
      also have ... ≤ 5 ^ 2 + 4 * 5 + 2
        using ⟨length ys ≤ 5⟩ by (intro add_mono power_mono) auto
      finally show T_slow_median ys ≤ 47 by simp
    qed
qed

```

also have $(\sum ys \leftarrow chop\ 5\ xs.\ 47) + length\ (chop\ 5\ xs) + 1 =$
 $48 * nat\ \lceil real\ n / 5 \rceil + 1$
by $(simp\ add:\ map_replicate_const\ length_chop)$
also have $\dots \leq 10 * n + 48$
by $linarith$
finally show $T_ms \leq 10 * n + 48$ **by** $simp$
qed

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

define T_rec1 **where**
 $T_rec1 = T_mom_select\ (((n + 4) \div 5 - 1) \div 2)\ (map\ slow_median\ (chop\ 5\ xs))$
from $False$ **have** $((length\ xs + 4) \div 5 - Suc\ 0) \div 2 < nat\ \lceil real\ (length\ xs) / 5 \rceil$
by $linarith$
hence $T_rec1 \leq T'_mom_select\ (length\ (map\ slow_median\ (chop\ 5\ xs)))$
using $False$ **unfolding** T_rec1_def **by** $(intro\ IH(1))\ (auto\ simp:\ length_chop)$
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 $\dots \leq T'_mom_select\ (length\ ls)$
by $(rule\ IH(2))\ (use\ \langle k < nl \rangle\ False\ in\ \langle auto\ simp:\ defs \rangle)$
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  $\dots \leq T'_mom\_select\ (length\ gs)$ 
proof (rule IH(3))
  show  $\neg n \leq 20$ 
    using False by auto
  show  $\neg k < nl \neg k < nl + ne$ 
    using  $\langle k \geq nl + ne \rangle$  by (auto simp: nl_def ne_def)
  have  $length\ xs = nl + ne + length\ gs$ 
    unfolding defs by (rule length_partition3) (simp_all add: partition3_def)
  thus  $k - nl - ne < length\ gs$ 
    using  $\langle k \geq nl + ne \rangle\ \langle k < length\ xs \rangle$  by (auto simp: nl_def ne_def)
qed
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  $T'_mom\_select\ (length\ gs) \leq T'_mom\_select\ (nat\ \lceil 0.7 * n$ 
 $+ 3 \rceil)$ 
    by (rule T'_mom_select_mono)
  finally show ?thesis .
qed

```

Now for the final inequality chain:

```

have  $T\_mom\_select\ k\ xs =$ 
  (if  $k < nl$  then  $T\_mom\_select\ k\ ls$ 
   else  $T\_length\ es +$ 
   (if  $k < nl + ne$  then  $0$  else  $T\_mom\_select\ (k - nl - ne)\ gs$ ))
+
   $T\_mom\_select\ (((n + 4) \div 5 - 1) \div 2)\ (map\ slow\_median$ 
 $(chop\ 5\ xs)) +$ 
   $T\_chop\ 5\ xs + T\_map\ T\_slow\_median\ (chop\ 5\ xs) + T\_partition3$ 
 $x\ xs +$ 
   $T\_length\ ls + T\_length\ xs + 1$  using False
by (subst T_mom_select_simps;

```

```

      unfold Let_def n_def [symmetric] x_def [symmetric] nl_def
[symmetric]
      ne_def [symmetric] prod.case tw [symmetric]) simp_all
    also have ... ≤ T_rec2 + T_rec1 + T_ms + 2 * n + nl + ne +
T_chop 5 xs + 5 using False
    by (auto simp add: T_rec1_def T_rec2_def T_partition3
      T_length T_ms_def nl_def ne_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 ≤ 10 * n + 48⟩
    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⌉) + 19 * n + 54
      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  $\text{nat } \lceil a*n+b \rceil < n$ 
  using  $\langle n > n0 \rangle$  by linarith
qed

lemma akra_bazzi_light_aux2:
  fixes  $f :: \text{nat} \Rightarrow \text{real}$ 
  fixes  $n_0 :: \text{nat}$  and  $a\ b\ c\ d :: \text{real}$  and  $C1\ C2\ C_1\ C_2 :: \text{real}$ 
  assumes bounds:  $a > 0\ c > 0\ a + c < 1\ C_1 \geq 0$ 
  assumes rec:  $\forall n > n_0. f\ n = f\ (\text{nat } \lceil a*n+b \rceil) + f\ (\text{nat } \lceil c*n+d \rceil) + C_1 * n + C_2$ 
  assumes ineqs:  $n_0 > (\max\ 0\ b + \max\ 0\ d + 2) / (1 - a - c)$ 
     $C_3 \geq C_1 / (1 - a - c)$ 
     $C_3 \geq (C_1 * n_0 + C_2 + C_4) / ((1 - a - c) * n_0 - \max\ 0\ b - \max\ 0\ d - 2)$ 
     $\forall n \leq n_0. f\ n \leq C_4$ 
  shows  $f\ n \leq C_3 * n + C_4$ 
proof (induction n rule: less_induct)
  case (less n)
  have  $0 \leq C_1 / (1 - a - c)$ 
    using bounds by auto
  also have  $\dots \leq C_3$ 
    by fact
  finally have  $C_3 \geq 0$  .

  show ?case
  proof (cases n > n0)
    case False
    hence  $f\ n \leq C_4$ 
      using ineqs(4) by auto
    also have  $\dots \leq C_3 * \text{real } n + C_4$ 
      using bounds  $\langle C_3 \geq 0 \rangle$  by auto
    finally show ?thesis .
  next
  case True
  have nonneg:  $a * n \geq 0\ c * n \geq 0$ 
    using bounds by simp_all

  have  $(\max\ 0\ b + 1) / (1 - a) \leq (\max\ 0\ b + \max\ 0\ d + 2) / (1 - a - c)$ 
    using bounds by (intro frac_le) auto
  hence  $n_0 \geq (\max\ 0\ b + 1) / (1 - a)$ 
    using ineqs(1) by linarith
  hence rec_less1:  $\text{nat } \lceil a*n+b \rceil < n$ 
    using bounds  $\langle n > n_0 \rangle$  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 n₀ ≥ (max 0 d + 1) / (1 - c)
      using ineqs(1) by linarith
    hence rec_less2: nat ⌈c*n+d⌉ < n
      using bounds ⟨n > n₀⟩ by (intro akra_bazzi_light_aux1[of _ n₀]) auto

    have f n = f (nat ⌈a*n+b⌉) + f (nat ⌈c*n+d⌉) + C₁ * n + C₂
      using ⟨n > n₀⟩ by (subst rec) auto
    also have ... ≤ (C₃ * nat ⌈a*n+b⌉ + C₄) + (C₃ * nat ⌈c*n+d⌉ +
C₄) + C₁ * n + C₂
      using rec_less1 rec_less2 by (intro add_mono less.IH) auto
    also have ... ≤ (C₃ * (a*n+max 0 b+1) + C₄) + (C₃ * (c*n+max 0
d+1) + C₄) + C₁ * n + C₂
      using bounds ⟨C₃ ≥ 0⟩ nonneg by (intro add_mono mult_left_mono
order.refl; linarith)
    also have ... = C₃ * n + ((C₃ * (max 0 b + max 0 d + 2) + 2 *
C₄ + C₂) -
      (C₃ * (1 - a - c) - C₁) * n)
      by (simp add: algebra_simps)
    also have ... ≤ C₃ * n + ((C₃ * (max 0 b + max 0 d + 2) + 2 *
C₄ + C₂) -
      (C₃ * (1 - a - c) - C₁) * n₀)
      using ⟨n > n₀⟩ ineqs(2) bounds
    by (intro add_mono diff_mono order.refl mult_left_mono) (auto simp:
field_simps)
    also have (C₃ * (max 0 b + max 0 d + 2) + 2 * C₄ + C₂) - (C₃ *
(1 - a - c) - C₁) * n₀ ≤ C₄
      using ineqs bounds by (simp add: field_simps)
    finally show f n ≤ C₃ * real n + C₄
      by (simp add: mult_right_mono)
  qed
qed

```

```

lemma akra_bazzi_light:
  fixes f :: nat ⇒ real
  fixes n₀ :: nat and a b c d C₁ C₂ :: real
  assumes bounds: a > 0 c > 0 a + c < 1 C₁ ≥ 0
  assumes rec: ∀ n > n₀. f n = f (nat ⌈a*n+b⌉) + f (nat ⌈c*n+d⌉) + C₁ *
n + C₂
  shows ∃ C₃ C₄. ∀ n. f n ≤ C₃ * real n + C₄
proof -

```

define n_0' **where** $n_0' = \max n_0 \text{ (nat } \lceil (\max 0 \ b + \max 0 \ d + 2) / (1 - a - c) + 1 \rceil)$
define C_4 **where** $C_4 = \text{Max } (f \text{ ' } \{..n_0'\})$
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 $\forall n > n_0'. f \ n = f \ (\text{nat } \lceil a*n+b \rceil) + f \ (\text{nat } \lceil c*n+d \rceil) + C_1 * n + C_2$
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 $(\max 0 \ b + \max 0 \ d + 2) / (1 - a - c) < \text{nat } \lceil (\max 0 \ b + \max 0 \ d + 2) / (1 - a - c) + 1 \rceil$
by *linarith*
also have $\dots \leq n_0'$
by (*simp add: n_0'_def*)
finally show $(\max 0 \ b + \max 0 \ d + 2) / (1 - a - c) < \text{real } n_0'.$
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 $f :: \text{nat} \Rightarrow \text{nat}$
fixes $n_0 :: \text{nat}$ **and** $a \ b \ c \ d :: \text{real}$ **and** $C_1 \ C_2 :: \text{nat}$
assumes *bounds*: $a > 0 \ c > 0 \ a + c < 1 \ C_1 \geq 0$
assumes *rec*: $\forall n > n_0. f \ n = f \ (\text{nat } \lceil a*n+b \rceil) + f \ (\text{nat } \lceil c*n+d \rceil) + C_1 * n + C_2$
shows $\exists C_3 \ C_4. \forall n. f \ n \leq C_3 * n + C_4$
proof –
have $\exists C_3 \ C_4. \forall n. \text{real } (f \ n) \leq C_3 * \text{real } n + C_4$
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*: $\forall n. \text{real } (f \ n) \leq C_3 * \text{real } n + C_4$
by *blast*
have $f \ n \leq \text{nat } \lceil C_3 \rceil * n + \text{nat } \lceil C_4 \rceil$ **for** n

```

proof –
  have  $\text{real } (f\ n) \leq C_3 * \text{real } n + C_4$ 
    using le by blast
  also have  $\dots \leq \text{real } (\text{nat } \lceil C_3 \rceil) * \text{real } n + \text{real } (\text{nat } \lceil C_4 \rceil)$ 
    by (intro add_mono mult_right_mono; linarith)
  also have  $\dots = \text{real } (\text{nat } \lceil C_3 \rceil * n + \text{nat } \lceil C_4 \rceil)$ 
    by simp
  finally show ?thesis by linarith
qed
thus ?thesis by blast
qed

lemma T'_mom_select_le':  $\exists C_1\ C_2. \forall n. T'_\text{mom\_select } n \leq C_1 * n + C_2$ 
proof (rule akra_bazzi_light_nat)
  show  $\forall n > 20. T'_\text{mom\_select } n = T'_\text{mom\_select } (\text{nat } \lceil 0.2 * n + 0 \rceil)$ 
  +
     $T'_\text{mom\_select } (\text{nat } \lceil 0.7 * n + 3 \rceil) + 19 * n + 54$ 
    using T'_mom_select.simps by auto
qed auto

end

```

50 Time Functions in Locales — An Example

```

theory Time_Locale_Example
imports
  HOL-Library.Time_Functions
  HOL-Library.AList
  Map_Specs
begin

```

If you want to reason about the time complexity of functions in a locale, you need to parameterize the locale with time functions for all functions that are utilized. More precisely, if you are in a locale parameterized by some function f and you define a function g that uses f , and you want to define T_g , it will depend on T_f , which you have to make an additional parameter of the locale. Only then will *time_fun* g work. Below we show a realistic example.

50.1 Basic Time Functions

```

time_fun AList.update
  time_fun needs uncurried defining equations

```

lemma

map_of_simps': $\text{map_of } [] (x :: 'a) = (\text{None} :: 'b \text{ option})$
 $\text{map_of } ((a :: 'a, b :: 'b) \# ps) x = (\text{if } x = a \text{ then } \text{Some } b \text{ else } \text{map_of } ps x)$
by *auto*

time_fun *map_of* **equations** *map_of_simps'*

lemma *T_map_ub*: $T_map_of\ ps\ a \leq \text{length } ps + 1$

by(*induction ps*) *auto*

lemma *T_update_ub*: $T_update\ a\ b\ ps \leq \text{length } ps + 1$

by(*induction ps*) *auto*

lemma *length_AList_update_ub*: $\text{length } (AList.update\ a\ b\ ps) \leq \text{length } ps + 1$

by(*induction ps*) *auto*

50.2 Locale

Counting the elements in a list by means of a map that associates elements with their multiplicities in the list, like a ‘histogram’. The locale is parameterized with the map ADT and the timing functions for *lookup* and *update*.

locale *Count_List* = *Map* **where** *update* = *update* **for** *update* :: $'a \Rightarrow \text{nat} \Rightarrow 'm \Rightarrow 'm +$

fixes *T_lookup* :: $'m \Rightarrow 'a \Rightarrow \text{nat}$

and *T_update* :: $'a \Rightarrow \text{nat} \Rightarrow 'm \Rightarrow \text{nat}$

begin

definition *lookup_nat* :: $'m \Rightarrow 'a \Rightarrow \text{nat}$ **where**

lookup_nat m x = (*case lookup m x of* *None* $\Rightarrow 0$ | *Some n* $\Rightarrow n$)

time_definition *lookup_nat*

fun *count* :: $'m \Rightarrow 'a \text{ list} \Rightarrow 'm$ **where**

count m [] = *m* |

count m (x # xs) = *count (update x (lookup_nat m x + 1) m) xs*

time_fun *count*

end

50.3 Interpretation

Interpretation of *Count_List* with association lists as maps.

```

lemma map_of_AList_update: map_of (AList.update a b ps) = ((map_of
ps)(a ↦ b))
by(induction ps) auto

```

```

lemma map_of_AList_delete: map_of (AList.delete a ps) = (map_of
ps)(a := None)
by(induction ps) auto

```

```

global_interpretation CL: Count_List
where empty = [] and lookup = map_of
and update = AList.update and delete = AList.delete and invar = λ_.
True
and T_lookup = T_map_of and T_update = T_update
defines CL_count = CL.count and CL_T_count = CL.T_count
proof (standard, goal_cases)
  case 1
    show ?case by (rule ext) simp
  next
    case (2 m a b)
    show ?case by (rule map_of_AList_update)
  next
    case (3 m a)
    show ?case by (rule map_of_AList_delete)
  next
    case 4
    show ?case by(rule TrueI)
  next
    case (5 m a b)
    show ?case by(rule TrueI)
  next
    case (6 m a)
    show ?case by(rule TrueI)
qed

```

50.4 Complexity Proof

```

lemma CL.T_count ps xs ≤ 2 * length xs * (length xs + length ps + 1)
+ 1
proof(induction xs arbitrary: ps)
  case Nil
    then show ?case by simp
  next
    case (Cons a xs)
    let ?lps' = length ps + 1

```

```

    let ?na' = CL.lookup_nat ps a + 1
    let ?ps' = AList.update a ?na' ps
    have CL_T_count ps (a # xs) =
      T_map_of ps a + T_update a ?na' ps + CL_T_count (AList.update
a ?na' ps) xs + 1
    by simp
    also have ... ≤ 2 * ?lps' + CL_T_count ?ps' xs + 1
    using T_map_ub T_update_ub add_mono by (fastforce simp: mult_2)
    also have ... ≤ 2 * ?lps' + 2 * length xs * (length xs + length ?ps' +
1) + 1 + 1
    using Cons.IH by (metis (no_types, lifting) add.assoc add_mono_thms_linordered_semiring(3)
nat_add_left_cancel_le)
    also have ... ≤ 2 * ?lps' + 2 * length xs * (length xs + ?lps' + 1) + 1
+ 1
    using length_AList_update_ub
    by (metis add_mono_thms_linordered_semiring(2) add_right_mono
mult_le_mono2)
    also have ... ≤ 2 * length (a # xs) * (length (a # xs) + length ps + 1)
+ 1
    by (auto simp: algebra_simps)
    finally show ?case .
qed

end

```

51 Bibliographic Notes

Red-black trees The insert function follows Okasaki [15]. The delete function in theory *RBTree* follows Kahrs [11, 12], an alternative delete function is given in theory *RBTree2*.

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 Technology, 1983.
- [6] C. A. Crane. *Linear Lists and Priority 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>.