Work Bounds for Strongly Joinable Balanced Binary Search Trees

Marco Haucke

September 24, 2026

Abstract

This entry formalises the complexity analysis of set operations on strongly joinable trees as defined by Blelloch et al. [4]. It is proved that union, intersection, and difference run in time $O(m \log(n/m+1))$ for input sets of size m and n where $m \leq n$.

Contents

1 Introduction

Binary search trees are one of the most fundamental data structures in computer science. Together with their traversals, they can be used to efficiently implement algorithms for managing ordered data. Balanced variants such as AVL trees [3], red-black trees [7] or weight-balanced trees [13] keep the height logarithmic in the size by maintaining a structural invariant, which enables simple and efficient implementations of pointwise operations. Operations that combine two trees are a different matter. Given two sets of sizes $m \leq n$, inserting the elements of the smaller set one by one takes time $\Theta(m \log n)$, and flattening both trees to lists, merging them and rebuilding a tree takes $O(m + n)$. Neither is satisfactory across the whole range of m and n.

The *join-based* approach, proposed by Adams [1], improves on this. Here, union, intersection and difference are all expressed through a single function `join l k r` that concatenates (i.e. "joins") two trees l and r and a key k into one balanced tree.

Blelloch, Ferizovic and Sun [4] put Adams' ideas on a general footing. They characterise a balancing scheme by a real-valued *rank* function together with a small set of rules relating rank, tree size, the shape and cost of `join`, and call schemes obeying these rules *strongly joinable*. For every strongly joinable scheme they show that union, intersection and difference of trees with sizes $m \leq n$ take $O\big(m \log(\frac{n}{m} + 1)\big)$ work, which is optimal in the comparison model [8]. Further, they prove that AVL trees, red-black trees and weight-balanced trees are strongly joinable.

This topic is not merely of theoretical interest. Adams' original code lives on in the SML/NJ Library [15] and in MIT/GNU Scheme [2], and the same join-based algorithms underlie the `Set` modules of OCaml's standard library [11] as well as Jane Street's *Base* [9], the immutable `TreeSet` of Scala [6], and the `Data.Set` implementation of Haskell's *containers* library [10]. Sun et al.'s PAM library [16, 17] shows the bound paying off in practice.

On the Isabelle side, the join-based approach is the subject of *Functional*

Data Structures and Algorithms [14, Chapter 10] and contained inside the `Set2_Join` locale in `HOL-Data_Structures`. Given a `join` that satisfies its functional specification, union, intersection and difference are defined generically and proven correct.

Verified counterparts exist in other proof assistants. The Rocq standard library's `MSetAVL` [19] is a port of OCaml's `Set` module, HOL4's `balanced_map` [18], which underlies the CakeML basis library, is a port of *containers*' `Data.Map`, and Breitner et al. [5] verify *containers*' `Data.Set` itself in Coq. All three establish functional correctness only. On the cost side, Li, Grodin and Harper [12] verify a precise bound for `join` on red-black trees in the *calf* framework, but leave the cost of the set operations to future work.

To the author's knowledge, this entry contains the first mechanised proof of the bound. It extends `Set2_Join` with a locale `StronglyJoinable` that adds the rank-based balance predicate and the rules of Blelloch et al., and proves within it that the running-time functions of union, intersection and difference are in $O\big(m \log(\frac{n}{m} + 1)\big)$ for input trees of sizes $m \leq n$.

Running times are measured by step-counting functions [14, Section 1.5] generated by the `time_fun` command. As proof of concept, the locale is instantiated with the AVL trees of `HOL-Data_Structures`, as well as red-black trees mostly as defined in `Set2_Join_RBT`.

2 Real Analysis Prerequisites

This first part collects the real-analysis groundwork for the complexity analysis of the joinable-tree operations, independent of trees and algorithms. It provides

- elementary logarithm estimates
- strict monotonicity of $x \cdot \log_2(1 + n/x)$
- closed forms and bounds for geometric and arithmetic-geometric series
- a finite, uniformly weighted instance of Jensen's inequality

theory *Analytical*
imports
HOL−Analysis.Convex
begin

2.1 Logarithm estimates

lemma *log_split*:
assumes $a \geq 0$ $b \geq 0$
shows $log\ 2\ (a * b + 1) \leq log\ 2\ (a + 1) + log\ 2\ (b + 1)$

```
proof −
  have a ∗ b + 1 > 0
   using assms by (metis zero_le_mult_iff abs_add_one_gt_zero abs_of_nonneg
add.commute)
  moreover have a ∗ b + 1 ≤ (a + 1) ∗ (b + 1)
    using assms by argo
  ultimately have log 2 (a ∗ b + 1) ≤ log 2 ((a + 1) ∗ (b + 1))
    using log_mono by auto
  then show ?thesis
    using assms log_mult_pos by force
qed

lemma log_2powr_plus_1_le:
  fixes x :: real
  assumes x ≥ 0
  shows log 2 (2 powr x + 1) ≤ 1 + x
proof −
  have 2 powr x + 1 ≤ 2 ∗ (2 powr x)
    using assms ge_one_powr_ge_zero by simp
  then have 2 powr x + 1 ≤ 2 powr (x + 1)
    by (simp add: add.commute powr_mult_base)
  then have log 2 (2 powr x + 1) ≤ log 2 (2 powr (x + 1))
    by (intro log_mono) (auto intro: add_pos_nonneg)
  also have ... = x + 1
    by (simp add: log_powr)
  finally show ?thesis
    by force
qed
```

2.2 Monotonicity of $x \cdot \log_2(1 + n/x)$

```
corollary ln_diff_less: 0 < x ⟹ 0 < y ⟹ x ≠ y ⟹ ln x − ln y < (x − y) /
y for
x :: real
using ln_eq_minus_one[of x/y] ln_diff_le[of x y]
by (fastforce simp: diff_divide_distrib ln_divide_pos)

lemma ln_mono_1:
  fixes n :: real
  assumes n_pos: n > 0
  defines f ≡ (λx::real. x ∗ ln (1 + n/x))
  shows   strict_mono_on {0<..} f
proof −
  have (f has_real_derivative ln (x + n) − ln x − n / (x + n)) (at x)
    if x > 0 for x
    unfolding f_def using ‹x>0› n_pos
    by(auto intro!: derivative_eq_intros) (force simp add: ln_div divide_simps)+
  moreover have 0 < ln (x + n) − ln x − n / (x + n) if x > 0 for x
    using ln_diff_less[of x x + n] ‹x>0› n_pos
```

```
    by auto
  ultimately have ∃ y. (f has_real_derivative y) (at x) ∧ 0 < y if x > 0 for x
    using ‹x>0› by blast
  then show ?thesis
     by (metis DERIV_pos_imp_increasing greaterThan_iff order.strict_trans2
strict_mono_onI)
qed

corollary log_mono_1:
   fixes n :: real
   assumes n_pos: n > 0
   defines f ≡ (λx::real. x * log 2 (1 + n/x))
   shows   strict_mono_on {0<..} f
using ln_mono_1[OF n_pos] unfolding f_def strict_mono_on_def log_def
  by (simp add: divide_less_eq)

corollary xlog_mono:
  fixes n x z :: real
  assumes n > 0 0 < x x ≤ z
  shows x * log 2 (n / x + 1) ≤ z * log 2 (n / z + 1)
proof −
  let ?f = λx::real. x * log 2 (1 + n / x)
  have ?f x ≤ ?f z
  proof (cases x = z)
    case False
    have strict_mono_on {0<..} ?f
      using assms(1) log_mono_1 by simp
    then show ?thesis
      using assms False strict_mono_onD[of {0<..} ?f x z] by auto
  qed simp
  then show ?thesis
    by argo
qed

corollary xlog_term_mono:
  fixes n x z k c :: real
  assumes 0 < x x ≤ z 0 < n 0 ≤ k 0 ≤ c
  shows x * (k + c * log 2 (n / x + 1)) ≤ z * (k + c * log 2 (n / z + 1))
proof −
  have k * x ≤ k * z
    using assms(2,4) by (rule mult_left_mono)
  moreover have c * (x * log 2 (n / x + 1)) ≤ c * (z * log 2 (n / z + 1))
    using xlog_mono[OF assms(3,1,2)] assms(5) by (rule mult_left_mono)
  ultimately show ?thesis
    by argo
qed
```

2.3 Geometric and arithmetic-geometric series

The following lemmas derive the limit $\sum_k k\, c^k = \frac{c}{(1-c)^2}$ together with summability for $|c| < 1$.

```
lemma arith_geometric_sums:
  fixes z :: real
  assumes z: norm z < 1
  shows   (λn. of_nat n * z ^ n) sums (z / (1 - z)^2)
proof -
  have (∑ n. of_nat (Suc n) * z ^ n) = 1 / (1 - z) ^ 2
  proof (rule DERIV_unique)
    have ((λz. ∑ n. 1 * z ^ n) has_field_derivative (∑ n. of_nat (Suc n) * z ^
n)) (at z)
      using termdiffs_strong'[of 1 λ_. 1 z] z by (simp add: diffs_def)
    also have ?this ⟷ ((λz. 1 / (1 - z)) has_field_derivative (∑ n. of_nat
(Suc n) * z ^ n)) (at z)
    proof (rule DERIV_cong_ev)
      have eventually (λx. x ∈ {x. dist 0 x < 1}) (nhds z)
        by (rule eventually_nhds_in_open, rule open_ball) (use z in auto)
      then show eventually (λx. (∑ n. 1 * x ^ n) = 1 / (1 - x)) (nhds z)
        by eventually_elim (auto simp: suminf_geometric)
    qed (auto simp: diffs_def)
    finally show ((λz. 1 / (1 - z)) has_field_derivative (∑ n. of_nat (Suc n) *
z ^ n)) (at z) .
  next
    show ((λz. 1 / (1 - z)) has_field_derivative (1 / (1 - z) ^ 2)) (at z)
          using z by (auto intro!: derivative_eq_intros simp: divide_simps
power2_eq_square)
  qed
  hence (λn. of_nat (Suc n) * z ^ n) sums (1 / (1 - z) ^ 2)
    using termdiff_converges[of z 1 λ_. 1] z by (simp add: diffs_def sums_iff)
  hence (λn. of_nat (Suc n) * z ^ n - z ^ n) sums (1 / (1 - z) ^ 2 - 1 / (1 -
z))
    using z by (intro sums_diff geometric_sums)
  also have (λn. of_nat (Suc n) * z ^ n - z ^ n) = (λn. of_nat n * z ^ n)
    by (simp add: algebra_simps)
  also have (1 / (1 - z) ^ 2 - 1 / (1 - z)) = z / (1 - z) ^ 2
    using assms by (simp add: divide_simps) (simp add: power2_eq_square alge-
bra_simps)
  finally show ?thesis .
qed

corollary arith_geometric_summable:
  fixes c :: real
  assumes norm c < 1
  shows   summable (λn. n * c^n)
using arith_geometric_sums assms summable_def by fastforce
```

The analysis eventually instantiates the series with the geometric ratio

$q(c_u) = 2^{-1/c_u}$, which lies strictly between 0 and 1 for $c_u > 0$. The constant definitions $S_1(c_u)$ and $S_2(c_u)$ bound the geometric and arithmetic-geometric series in this ratio over any finite index set.

```
lemma powr_split_nat:
  fixes a :: real and cu :: real and i :: nat
  assumes a > 0
  shows a powr (real i / cu) = (a powr (1 / cu)) ^ i
   by (metis assms(1) div_by_1 divide_divide_eq_right powr_eq_0_iff
    powr_eq_one_iff_gen powr_power times_divide_eq_right zero_neq_one)

definition geom_ratio where geom_ratio cu = 1 / (2 powr (1 / cu))

lemma geom_ratio_range:
  fixes cu :: real
  assumes cu_pos:cu > 0
  shows 0 < geom_ratio cu ∧ geom_ratio cu < 1
using assms by (simp add: geom_ratio_def)

lemma inv_powr_geom_ratio:
  fixes cu :: real and i :: nat
  assumes cu > 0
  shows 1 / (2 powr (real i / cu)) = geom_ratio cu ^ i
by (simp add: power_one_over powr_split_nat geom_ratio_def)

definition S1 where S1 cu = (2 powr (1 / cu)) / ((2 powr (1 / cu)) − 1)
definition S2 where S2 cu = (geom_ratio cu) / (1 − geom_ratio cu)^2

lemma geom_ratio_sum_le:
  fixes cu :: real
  assumes cu > 0 finite A
  shows (∑ i∈A. (geom_ratio cu) ^ i) ≤ S1 cu
proof −
  have qr: 0 < geom_ratio cu ∧ geom_ratio cu < 1
    using geom_ratio_range[OF assms(1)] .
  have (∑ i∈A. (geom_ratio cu) ^ i) < 1 / (1 − geom_ratio cu)
    by (metis qr assms(2) geometric_sum_less)
  also have 1 / (1 − geom_ratio cu) = S1 cu
    using qr by (simp add: S1_def geom_ratio_def field_simps)
  finally show ?thesis
    by force
qed

lemma geom_ratio_arith_sum_le:
  assumes cu > 0 finite A
  shows (∑ i∈A. real i ∗ (geom_ratio cu) ^ i) ≤ S2 cu
proof −
  have qr:0 < geom_ratio cu ∧ geom_ratio cu < 1
    using geom_ratio_range[OF assms(1)] .
  have summable (λi::nat. real i ∗ (geom_ratio cu)^i)
```

```
    using qr arith_geometric_summable by fastforce
  moreover have 0 ≤ (real i * (geom_ratio cu)^i) for i
    using qr by simp
   ultimately have (∑i∈A. real i * (geom_ratio cu)^i) ≤ (∑i. real i *
(geom_ratio cu)^i)
    using assms(2) sum_le_suminf summable by blast
  also have ... = S2 cu
  proof −
     have (λi::nat. real i * (geom_ratio cu)^i) sums ((geom_ratio cu) / (1 −
geom_ratio cu)^2)
      using qr by (simp add: arith_geometric_sums)
    then show ?thesis
      by (metis S2_def sums_unique)
  qed
  finally show ?thesis .
qed
```

Both instantiations below use $c_u = 2$, for which the series constants have closed forms in $\sqrt{2}$.

```
lemma sqrt2_bounds: sqrt 2 * sqrt 2 = 2 1 < sqrt 2 sqrt 2 ≤ 1.415
proof −
  show sqrt 2 * sqrt 2 = 2
    by fastforce
  show 1 < sqrt 2
    by simp
  show sqrt 2 ≤ 1.415
    by (rule real_le_lsqrt) (simp_all add: power2_eq_square)
qed

lemma S1_two: S1 2 = 2 + sqrt 2
proof −
  have S1 2 = sqrt 2 / (sqrt 2 − 1)
    by (simp add: S1_def powr_half_sqrt)
  also have ... = 2 + sqrt 2
    using sqrt2_bounds(1,2) by (simp add: field_simps)
  finally show ?thesis .
qed

lemma S2_two: S2 2 = 4 + 3 * sqrt 2
proof −
  have geom_ratio 2 = 1 / sqrt 2
    by (simp add: geom_ratio_def powr_half_sqrt)
  then have S2 2 = (1 / sqrt 2) / (1 − 1 / sqrt 2)^2
    by (simp add: S2_def)
  also have ... = 2 / (3 * sqrt 2 − 4)
    using sqrt2_bounds(1,2) by (simp add: power2_eq_square field_simps)
  also have ... = 4 + 3 * sqrt 2
  proof −
    have 4 / 3 < sqrt 2
```

by (*rule real_less_rsqrt*) (*simp add*: *power2_eq_square*)
then show *?thesis*
using *sqrt2_bounds*(*1*) **by** (*simp add*: *field_simps*)
qed
finally show *?thesis* **.**
qed

2.4 Jensen's inequality for finite sets

Jensen's inequality relates the value of a concave function at a weighted average to the average of its values. The below is a special case of *concave_on_sum*.

corollary *jensen_finite*:
fixes $S :: 'b\ set$
defines $a \equiv (\lambda i::'b.\ 1\ /\ card\ S)$
assumes *finite S* $S \neq \{\}$
and *concave_on D f*
and $\bigwedge i.\ i \in S \Longrightarrow x\ i \in D$
shows $f\ ((1\ /\ card\ S) *_R (\sum i{\in}S.\ x\ i))$
$\geq (1\ /\ card\ S) * (\sum i{\in}S.\ f\ (x\ i))$
proof −
have *weights*: $\bigwedge i.\ i \in S \Longrightarrow a\ i \geq 0\ (\sum i{\in}S.\ a\ i) = 1$
unfolding *a_def* **using** *assms* **by** *auto*
show *?thesis*
using *concave_on_sum*[*OF assms*(*2*,*3*,*4*) *weights*(*2*) *weights*(*1*) *assms*(*5*)]
unfolding *a_def* **by** (*simp add*: *sum_distrib_left scaleR_right.sum*)
qed

corollary *jensen_list*:
assumes $xs \neq []$
and *concave_on C f*
and $\bigwedge x.\ x \in set\ xs \Longrightarrow x \in C$
shows $f\ ((1\ /\ length\ xs) *_R sum\_list\ xs)$
$\geq (1\ /\ length\ xs) * sum\_list\ (map\ f\ xs)$
proof −
let *?S* = {..<*length xs*}
let *?x* = $\lambda i.\ xs\ !\ i$

have $f\ ((1\ /\ card\ ?S) *_R (\sum i{\in}?S.\ xs\ !\ i)) \geq (1\ /\ card\ ?S) * (\sum i{\in}?S.\ f\ (xs\ !\ i))$
proof −
have *finite ?S*
by *simp*
moreover have $?S \neq \{\}$
using *assms*(*1*) **by** *blast*
moreover have *in_C*: $\bigwedge i.\ i \in ?S \Longrightarrow xs\ !\ i \in C$
by (*simp add*: *assms*(*3*))
ultimately show *?thesis* **using** *assms jensen_finite* **by** *blast*
qed

moreover have *card ?S = length xs*
by *simp*
moreover have *sum_list xs* = $(\sum i{\in}?S.\ xs\ !\ i)$
by (*simp add*: *lessThan_atLeast0 sum_list_sum_nth*)
moreover have *sum_list* (*map f xs*) = $(\sum i{\in}?S.\ f\ (xs\ !\ i))$
by (*metis* (*mono_tags*, *lifting*) *length_map lessThan_atLeast0*
lessThan_iff nth_map sum.cong sum.list_conv_set_nth)
ultimately show *?thesis*
by *force*
qed

end

3 Balanced Binary Trees

theory *BalancedBinary*
imports
HOL−Library.Tree
Complex_Main
begin

BalancedShape captures the *shape* of a balancing scheme i.e. a rank function together with the constants c_l and c_u. The *balanced* predicate is defined relative to these, the locale then captures the emergent properties of the scheme as defined by [4]. Here, $c_l > 0$ is demanded explicitly. With $c_l = 0$ the rank would never have to shrink towards the leaves, and the [4] itself divides by c_l in its Property 7. The actual guarantee that a scheme keeps its trees balanced is added by the *BalancedTree* locale further below, which connects an invariant to *balanced*.

locale *BalancedShape* =
fixes *rank* :: (*'a* ∗ *'b*) *tree* ⇒ *real*
fixes c_l c_u :: *real*
assumes *c_vals*:$c_l > 0 \wedge c_l \leq 1 \wedge c_u \geq 1$
assumes *rule_empty*: *rank Leaf* = *0*
begin

lemma *c_l_pos*: $0 < c_l$
using *c_vals* **by** *blast*

lemma *c_l_le_one*: $c_l \leq 1$
using *c_vals* **by** *blast*

lemma *c_u_ge_one*: $1 \leq c_u$
using *c_vals* **by** *blast*

corollary *c_l_nonneg*: $0 \leq c_l$
using *c_l_pos* **by** *linarith*

corollary *c_u_pos*: $0 < c_u$
using *c_u_ge_one* **by** *linarith*

corollary *c_u_nonneg*: $0 \leq c_u$
using *c_u_pos* **by** *linarith*

3.1 Balance Predicate

The central balance predicate is defined as a recursive function over the structure of the tree. For each node it ensures that left and right subtrees of any given parent node differ only by the balancing constants c_l and c_u in terms of *rank* relative to it.

fun *balanced* :: $('a * 'b)$ *tree* $\Rightarrow$ *bool* **where**
balanced Leaf = *True* |
balanced (*Node l a r*) =
(*max* (*rank l*) (*rank r*) + $c_l \leq$ *rank* (*Node l a r*) $\wedge$
min (*rank l*) (*rank r*) + $c_u \geq$ *rank* (*Node l a r*) $\wedge$
balanced l $\wedge$ *balanced r*)

It follows immediately that the ranks of the left/right subtrees of a parent node can differ at most by a constant, namely $c_\delta = c_u - c_l$ [4, Property 4].

definition $c_\delta = c_u - c_l$

lemma *dmax*:*balanced* (*Node l a r*) $\Longrightarrow |$*rank l* $-$ *rank r*$| \leq c_\delta$
using *c_δ_def* **by** *force*

lemma *balanced_children_explD*:
assumes *balanced* (*Node l a r*)
shows *rank l* $\leq$ *rank* (*Node l a r*) $- c_l$
and *rank r* $\leq$ *rank* (*Node l a r*) $- c_l$
and *rank* (*Node l a r*) $- c_u \leq$ *rank l*
and *rank* (*Node l a r*) $- c_u \leq$ *rank r*
using *assms* **by** *fastforce+*

3.2 Relationship between tree rank and height

Rank is bounded below by $c_l \cdot$ *height t* which is the direction of [4, Property 1] needed for the analysis.

lemma *rank_lower_height*:*balanced t* $\Longrightarrow c_l *$ *height t* $\leq$ *rank t*
proof(*induction t*)
case *Leaf*
then show *?case*
by (*simp add*: *rule_empty*)
next
case (*Node l a r*)

then have $c_l * height\ l \leq rank\ l$
by *simp*
moreover have $c_l * height\ r \leq rank\ r$
using *Node* **by** *simp*
ultimately have $c_l * max\ (height\ l)\ (height\ r) \leq max\ (rank\ l)\ (rank\ r)$
by (*simp add*: *mult_left_mono c_vals max_def*)
then have $c_l * max\ (height\ l)\ (height\ r) + c_l \leq max\ (rank\ l)\ (rank\ r) + c_l$
by *linarith*
also have $c_l * max\ (height\ l)\ (height\ r) + c_l = c_l * (max\ (height\ l)\ (height\ r) + 1)$
by (*simp add*: *distrib_left*)
finally have $c_l * (max\ (height\ l)\ (height\ r) + 1) \leq max\ (rank\ l)\ (rank\ r) + c_l$
by *linarith*
moreover have $c_l * height\ (Node\ l\ a\ r) \leq c_l * (max\ (height\ l)\ (height\ r) + 1)$
using *c_vals* **by** *simp*
ultimately show *?case*
using *Node* **by** *auto*
qed

corollary *rank_lower_height_inverse*:*balanced t* $\Longrightarrow$ *height t* $\leq$ *rank t* / c_l
by (*metis c_vals rank_lower_height pos_le_divide_eq mult_of_nat_commute*)

Which also implies *rank* is never negative.

corollary *rank_pos*:*balanced t* $\Longrightarrow$ *rank t* $\geq$ *0*
by (*metis rank_lower_height c_vals of_nat_0_le_iff mult_nonneg_nonneg order_trans order_less_le*)

The upper bound for *rank* holds as well and is added for sake of completeness.

lemma *rank_upper_height*:*balanced t* $\Longrightarrow$ *rank t* $\leq c_u *$ *height t*
proof(*induction t*)
case *Leaf*
then show *?case*
by (*simp add*: *rule_empty*)
next
case (*Node l a r*)
then have *rank l* $\leq c_u *$ *height l*
by *simp*
moreover have *rank r* $\leq c_u *$ *height r*
using *Node* **by** *simp*
ultimately have *max* (*rank l*) (*rank r*) $\leq c_u *$ *max* (*height l*) (*height r*)
using *c_vals* **by** (*auto simp*: *max_def intro*: *mult_left_mono order_trans*)
then have *max* (*rank l*) (*rank r*) + $c_u \leq c_u *$ *max* (*height l*) (*height r*) + c_u
using *c_vals* **by** *argo*
also have ... $\leq c_u * (max\ (height\ l)\ (height\ r) + 1)$
using *c_vals* **by** (*simp add*: *distrib_left*)
also have ... $\leq c_u *$ *height* (*Node l a r*)
using *c_vals* **by** *auto*
finally show *?case*

using *Node* **by** *auto*
qed

3.3 Relationship between tree rank and size

Size grows exponentially in rank [4, Property 2].

lemma *pow_shift*:
assumes $c_u \neq 0$
shows $2 * 2\ powr\ ((r - c_u)\ /\ c_u) = 2\ powr\ (r\ /\ c_u)$
proof −
have $2 * 2\ powr\ ((r - c_u)\ /\ c_u) = (2\ powr\ 1) * 2\ powr\ ((r - c_u)\ /\ c_u)$
by *simp*
also have $... = 2\ powr\ (1 + (r - c_u)\ /\ c_u)$
by (*simp add*: *powr_add*)
also have $1 + (r - c_u)\ /\ c_u = r\ /\ c_u$
using *assms* **by** (*simp add*: *field_simps*)
finally show *?thesis*
by *simp*
qed

lemma *size1_ge_powr_rank*:
$balanced\ t \Longrightarrow size1\ t \geq 2\ powr\ (rank\ t\ /\ c_u)$
proof (*induction t*)
case *Leaf*
show *?case*
by (*simp add*: *rule_empty*)
next
case (*Node l a r*)
then have
IHl: $size1\ l \geq 2\ powr\ (rank\ l\ /\ c_u)$
and
IHr: $size1\ r \geq 2\ powr\ (rank\ r\ /\ c_u)$
by *auto*

have *lower_l*:
$size1\ l \geq 2\ powr\ ((rank\ (Node\ l\ a\ r) - c_u)\ /\ c_u)$
proof −
have $rank\ (Node\ l\ a\ r) - c_u \leq rank\ l$
using *Node.prems* **by** *auto*
then have $2\ powr\ ((rank\ (Node\ l\ a\ r) - c_u)\ /\ c_u) \leq 2\ powr\ (rank\ l\ /\ c_u)$
using *c_vals divide_right_mono* **by** *fastforce*
then show *?thesis*
using *IHl* **by** *argo*
qed

have *lower_r*:
$size1\ r \geq 2\ powr\ ((rank\ (Node\ l\ a\ r) - c_u)\ /\ c_u)$
proof −
have $rank\ (Node\ l\ a\ r) - c_u \leq rank\ r$

```
    using Node.prems by auto
  then have 2 powr ((rank (Node l a r) − c_u) / c_u) ≤ 2 powr (rank r / c_u)
    using c_vals divide_right_mono by fastforce
  then show ?thesis
    using IHr by argo
qed

have real (size1 (Node l a r)) = real (size1 l) + real (size1 r)
  by simp
also have ... ≥ 2 ∗ (2 powr ((rank (Node l a r) − c_u) / c_u))
  using lower_l lower_r by linarith
finally show ?case
  using pow_shift by fastforce
qed

corollary rank_le_cu_log_size1:
  assumes balanced t
  shows rank t ≤ c_u ∗ log 2 (size1 t)
proof −
  have size1 t ≥ 2 powr (rank t / c_u)
    using size1_ge_powr_rank[OF assms] .
  then have log 2 (size1 t) ≥ log 2 (2 powr (rank t / c_u))
    using le_log_iff by auto
  then show ?thesis
    using c_vals by (simp add: field_simps)
qed
```

3.4 Layers and rank-roots

Layer i of a tree collects all nodes whose rank falls into $[i, i+1)$ i.e. its rank lies in the unit band $[i, i+1)$ [4, Definition 3]. In essence, *layer* discretizes the real valued *rank* into finite layers, enabling the summation arguments used in the asymptotic analysis.

```
abbreviation in_band :: nat ⇒ real ⇒ bool where
  in_band i x ≡ real i ≤ x ∧ x < real i + 1

fun layer :: ('a ∗ 'b) tree ⇒ nat ⇒ (('a ∗ 'b) tree) list where
layer Leaf _ = [] |
layer (Node l x r) i =
    ((if in_band i (rank (Node l x r)) then [(Node l x r)] else []) @
   layer l i
   @ layer r i)

lemma layer_empty_if_rank_lt:
  assumes Bt: balanced t
  assumes lt: rank t < i
  shows layer t i = []
using Bt lt c_vals by(induction t) auto
```

Where *layer* recurses the entire tree, *rank_roots* stops at the first node whose rank falls into layer i, which is called the *rank root* of that layer [4, Definition 4]. A layer then decomposes as the concatenation of sub-layers rooted at these nodes (cf. [4, Definition 5]).

```
fun rank_roots :: ('a * 'b) tree ⇒ nat ⇒ (('a * 'b) tree) list where
rank_roots Leaf _ = [] |
rank_roots (Node l x r) i =
    (if in_band i (rank (Node l x r)) then [(Node l x r)] else
   rank_roots l i
   @ rank_roots r i)
```

```
lemma rank_rank_roots: r ∈ set(rank_roots t i) ⟹ in_band i (rank r)
  by(induction t) (auto split: if_splits)
```

Balance is inherited.

```
lemma balanced_rank_roots: ⟦balanced t; r ∈ set(rank_roots t i)⟧ ⟹ balanced r
  by(induction t) (auto split: if_splits)
```

```
lemma layer_eq_concat_rank_roots:
  layer T i = concat (map (λr. layer r i) (rank_roots T i))
proof (induction T)
  case Leaf
  then show ?case by simp
next
  case (Node l x r)
  then show ?case
    by (cases in_band i (rank (Node l x r))) auto
qed
```

3.5 Size bound on layer and root nodes

The work bounds for union, intersection and difference will ultimately be obtained by decomposing the total work into layers and bounding each layer's contribution separately. Two main ingredients make this work:

- *Across* layers, the number of rank roots decays geometrically i.e. layer i has at most *size1* $t/2^{i/c_u}$ rank roots. This sharpens [4, Lemma 1], where the exponent is $i/(1 + c_u)$.
- *Within* a layer, the number of nodes belonging to each rank root is bounded by the constant $C = 2^{\lceil 1/c_l \rceil} - 1$. This absorbs intra-layer overhead into a constant factor [4, Property 7]

Both facts are established in this subsection and are consumed separately by the main theory, as the decay bound on rank roots and the constant bound per rank root. The combined geometric layer bound closing this subsection is not needed downstream, but is still shown to make it clear

that it is an emergent property of balanced trees. First, each rank root in layer i has rank at least i, so the exponential size lower bound applies.

corollary *rank_roots_size_lower*:
assumes *balanced t*
shows $r \in$ *set* (*rank_roots t i*) $\Longrightarrow$ *size1 r* $\geq$ *2 powr* (*i* / c_u)
proof −
fix *r*
assume *r_in*: $r \in$ *set* (*rank_roots t i*)

have $i \leq$ *rank r*
using *rank_rank_roots*[*OF r_in*] **by** *auto*
then have (*2*::*real*) *powr* (*real i* / c_u) $\leq$ (*2*::*real*) *powr* (*rank r* / c_u)
using *c_vals* **by**(*simp add*: *divide_simps*)
then show *size1 r* $\geq$ *2 powr* (*real i* / c_u)
using *size1_ge_powr_rank*[*OF balanced_rank_roots*[*OF assms r_in*]] **by** *linarith*
qed

Further, the combined size of all rank roots in a layer never exceeds the size of the whole tree.

lemma *rank_roots_size_sum*:
sum_list (*map size1* (*rank_roots t i*)) $\leq$ *size1 t*
by(*induction t*) *auto*

Combining both gives the desired bound on the number of rank roots.

lemma *rank_roots_count_powr*:
assumes *balanced t*
shows *length* (*rank_roots t i*) $*$ *2 powr* (*i* / c_u) $\leq$ *size1 t*
proof −
have *length* (*rank_roots t i*) $*$ *2 powr* (*i* / c_u)
$=$ *sum_list* (*map* (λ_. *2 powr* (*i* / c_u)) (*rank_roots t i*))
by (*simp add*: *sum_list_triv*)
also have ... $\leq$ *sum_list* (*map* (*real o size1*) (*rank_roots t i*))
using *rank_roots_size_lower*[*OF assms*] **by** (*simp add*: *sum_list_mono*)
also have ... $\leq$ *real* (*size1 t*)
using *rank_roots_size_sum*[*of t i*]
by (*metis list.map_comp of_nat_le_iff sum_list_of_nat*)
finally show *?thesis* **.**
qed

Restated via division.

corollary *rank_roots_count_powr_le*:
assumes *balanced t*
shows *length* (*rank_roots t i*)
$\leq$ *size1 t* / (*2 powr* (*i* / c_u))
proof −
have *length* (*rank_roots t i*) $*$ *2 powr* (*i* / c_u) $\leq$ *size1 t*
using *rank_roots_count_powr*[*OF assms*] **.**
moreover have *0* $<$ *2 powr* (*real i* / c_u)

```
    by simp
  ultimately show ?thesis
    by (simp add: pos_le_divide_eq)
qed
```

A balanced tree whose rank overshoots layer i by at most d has at most $2^{\lceil \frac{d}{c_l} \rceil} - 1$ nodes in that layer. The budget d shrinks by c_l at every level, giving the exponential bound. The parameter d is generalized beyond the natural choice $d = 1$ because the induction step requires applying the lemma to children with a reduced budget.

```
lemma layer_len_budget:
  fixes d :: real
  assumes balanced t
  assumes rank t < real i + d
  assumes 0 ≤ d
  shows length (layer t i) ≤ 2^(nat (ceiling (d / c_l))) - 1
using assms
proof (induction t arbitrary: d)
  case Leaf
  then show ?case
    by simp
next
  case (Node l x r)

  let ?t = Node l x r
  let ?m = nat (ceiling (d / c_l))

  have Bl: balanced l and Br: balanced r
    using Node by auto

  show ?case
  proof (cases ?m = 0)
    case True
    then show ?thesis
     using layer_empty_if_rank_lt[OF Node.prems(1)] Node.prems(2,3) c_l_pos

      by (auto simp add: divide_le_0_iff)
  next
    case False
    have mpos: ?m ≥ 1
      using False by linarith
    have child_cut_l: rank l < real i + (d - c_l)
      using Node.prems(1,2) by force
    have child_cut_r: rank r < real i + (d - c_l)
      using Node.prems(1,2) by force
    have ceil_step:
      nat (ceiling ((d - c_l) / c_l)) ≤ ?m - 1
      using c_vals by (auto simp: diff_divide_distrib)
```

```
    have length (layer ?t i) ≤ 1 + length (layer l i) + length (layer r i)
      by simp
    also have ... ≤ 1 + (2^(?m − 1) − 1) + (2^(?m − 1) − 1)
    proof −
      have length (layer l i) ≤ 2^(?m − 1) − 1
      proof −
        have length (layer l i) ≤ 2^(nat (ceiling ((d − c_l) / c_l))) − 1
          using Bl Node.IH(1) child_cut_l layer_empty_if_rank_lt by force
        also have ... ≤ 2^(?m − 1) − 1
              using ceil_step by (metis diff_le_mono le_add_same_cancel1
one_add_one power_increasing zero_le_one)
        finally show ?thesis .
      qed

      moreover have length (layer r i) ≤ 2^(?m − 1) − 1
      proof −
        have length (layer r i) ≤ 2^(nat (ceiling ((d − c_l) / c_l))) − 1
          using Br Node.IH(2) child_cut_r layer_empty_if_rank_lt by force
        also have ... ≤ 2^(?m − 1) − 1
              using ceil_step by (metis diff_le_mono le_add_same_cancel1
one_add_one power_increasing zero_le_one)
        finally show ?thesis .
      qed

      ultimately show ?thesis by linarith
    qed
    also have ... = 2 * 2^(?m − 1) − 1
        using one_le_power by fastforce
    also have ... = 2^?m − 1
      using mpos False by (metis power_eq_if)
    finally show ?thesis .
  qed
qed
```

Instantiating with $d = 1$ begets the actual constant.

```
definition C :: nat where C = 2^(nat (ceiling (1 / c_l))) − 1

lemma C_pos: real C ≥ 0
  by force

corollary layer_bound_const:
  assumes balanced t rank t < real i + 1
  shows length (layer t i) ≤ C
  using layer_len_budget[OF assms(1) assms(2)] C_def by simp
```

Thus the total number of nodes in layer i is at most C times the number of rank roots.

```
lemma length_concat_map_le_sum:
```

```
  length (concat (map f xs)) ≤ sum_list (map (λx. length (f x)) xs)
  by (induction xs) auto

lemma layer_bound_via_rank_roots:
  assumes BT: balanced T
  shows length (layer T i)
      ≤ C * length (rank_roots T i)
proof −
  have decomp: layer T i = concat (map (λr. layer r i) (rank_roots T i))
    using layer_eq_concat_rank_roots .
  then have length (layer T i)
     = length (concat (map (λr. layer r i) (rank_roots T i)))
    by simp
  also have ... ≤ sum_list (map (λr. length (layer r i)) (rank_roots T i))
    using length_concat_map_le_sum by metis
  also have ... ≤ sum_list (map (λ_. C) (rank_roots T i))
      using balanced_rank_roots[OF BT] rank_rank_roots layer_bound_const
C_def by (meson sum_list_mono)
  also have sum_list (map (λ_. C) (rank_roots T i)) = C * (length (rank_roots
T i))
    by (simp add: sum_list_triv)
  finally show ?thesis
   by simp
qed
```

Together, this yields the fully combined geometric layer bound.

```
corollary layer_bound_geometric:
  assumes balanced t
  shows (length (layer t i)) ≤ C * (size1 t / (2 powr (i / c_u)))
proof −
  have
    length (layer t i) ≤ C * length (rank_roots t i)
    using layer_bound_via_rank_roots[OF assms] .
  also have
     ... ≤ C * (size1 t / (2 powr (i / c_u)))
       using rank_roots_count_powr_le[OF assms] by (metis mult_left_mono
of_nat_0_le_iff of_nat_mult)
  finally show ?thesis
    by simp
qed

end
```

3.6 Balanced schemes

A *BalancedTree* is a *BalancedShape* together with an invariant that implies balance. This records the balancing rule of [4].

```
locale BalancedTree = BalancedShape rank c_l c_u
  for rank :: ('a × 'b) tree ⇒ real
```

and c_l c_u :: *real*
+
fixes *inv* :: $('a \times 'b)$ *tree* $\Rightarrow$ *bool*
assumes *rule_bal*: *inv t* $\Longrightarrow$ *balanced t*
begin

declare *rule_bal*[*intro*]

corollary *rank_pos'*[*simp*]: *inv t* $\Longrightarrow$ *0* $\leq$ *rank t*
using *rank_pos rule_bal* **by** *blast*

corollary *dmax'*: *inv* (*Node l a r*) $\Longrightarrow$ $|rank\ l - rank\ r| \leq c_\delta$
using *dmax rule_bal* **by** *blast*

corollary *rank_le_cu_log_size1'*: *inv t* $\Longrightarrow$ *rank t* $\leq c_u *$ *log 2* (*size1 t*)
using *rank_le_cu_log_size1 rule_bal* **by** *blast*

corollary *rank_lower_height_inverse'*: *inv t* $\Longrightarrow$ *height t* $\leq$ *rank t* / c_l
using *rank_lower_height_inverse rule_bal* **by** *blast*

corollary *inv_children_explD*:
assumes *inv* (*Node l a r*)
shows *rank l* $\leq$ *rank* (*Node l a r*) $- c_l$
and *rank r* $\leq$ *rank* (*Node l a r*) $- c_l$
and *rank* (*Node l a r*) $- c_u \leq$ *rank l*
and *rank* (*Node l a r*) $- c_u \leq$ *rank r*
using *balanced_children_explD*[*OF rule_bal*[*OF assms*]] **by** *blast+*
end

end

4 Strongly Joinable Trees

theory *StronglyJoinable*
imports
Analytical
BalancedBinary
HOL−Data_Structures.Set2_Join
HOL−Library.Time_Commands
HOL−Library.Landau_Symbols
HOL−Library.Going_To_Filter
begin

The *StronglyJoinable* locale combines the functional correctness setting of *Set2_Join* with the balancing scheme of *BalancedTree* and adds the remaining strongly joinable rules of [4]:

- *rule_mono*: the rank of a join dominates the ranks of both arguments

- *rule_sub*: replacing the subtrees of a node by trees whose rank exceeds the original by at most x yields a join whose rank exceeds the rank of the original node by at most x
- *rule_cost*: the cost of a join is linear in the rank difference of its arguments

The submodularity rule *rule_sub* deviates from the definition in [4]. The reason is support for flag-free *join* implementations of the type fixed by *Set2_Join* and the price is a larger constant in the lower bound on the rank of union. See the appendix for details.

Finally, the added sanity rule *join_size* demands that *join* is not degenerate i.e. it adds exactly the pivot element and nothing else. Under *bst* preconditions this is already implied by functional correctness, but stating it directly avoids threading the *bst* assumption through the entire work analysis. It trivially holds for any correct implementation of *join*.

Note that for *rule_cost* there is no guarantee that *T_join* is the actual timing function corresponding to *join*. The analysis is therefore stated relative to the function supplied by an instantiation.

The final bounds are stated in terms of the smaller size m and the larger size n of the two inputs.

abbreviation *size_min* :: *'a tree* ⇒ *'a tree* ⇒ *real* **where**
  size_min ta tb ≡ *real* (*min* (*size1 ta*) (*size1 tb*))
abbreviation *size_max* :: *'a tree* ⇒ *'a tree* ⇒ *real* **where**
  size_max ta tb ≡ *real* (*max* (*size1 ta*) (*size1 tb*))

locale *StronglyJoinable* =
  Set2_Join join inv +
  BalancedTree rank c_l c_u *inv*
  for *rank* :: (*'a*::*linorder* ∗ *'b*) *tree* ⇒ *real*
  and c_l c_u :: *real*
  and *join* :: (*'a*∗*'b*) *tree* ⇒ *'a* ⇒ (*'a*∗*'b*) *tree* ⇒ (*'a*∗*'b*) *tree*
  and *inv* :: (*'a*∗*'b*) *tree* ⇒ *bool*
  +
  fixes *T_join* :: (*'a*∗*'b*) *tree* ⇒ *'a* ⇒ (*'a*∗*'b*) *tree* ⇒ *nat*
  and k_0 k_1 :: *real*
  assumes *rule_mono*:
    ⟦*inv l*; *inv r*⟧ ⟹ *max* (*rank l*) (*rank r*) ≤ *rank* (*join l a r*)
  assumes *rule_sub*:
    ⟦ *rank l'* ≤ *rank l* + *x*; *rank r'* ≤ *rank r* + *x*;
      inv l'; *inv r'*; *inv* (*Node l* (*a*,*b*) *r*) ⟧ ⟹
      rank (*join l' a r'*) ≤ *rank* (*Node l* (*a*,*b*) *r*) + *x*
  assumes *join_size*: ⟦*inv l*; *inv r*⟧ ⟹ *size* (*join l a r*) = *size l* + *size r* + *1*
  assumes *rule_cost*:
    ⟦*inv l*; *inv r*⟧ ⟹ *real* (*T_join l a r*) ≤ k_0 + k_1 ∗ |*rank l* − *rank r*|
  assumes k_0*_nonneg*: *0* ≤ k_0
  assumes k_1*_nonneg*: *0* ≤ k_1

begin

lemma *split_invL*[*dest*]: ⟦*split x t* = (*l*,*b*,*r*); *inv t*⟧ ⟹ *inv l*
using *split_inv* **by** *blast*

lemma *split_invR*[*dest*]: ⟦*split x t* = (*l*,*b*,*r*); *inv t*⟧ ⟹ *inv r*
using *split_inv* **by** *blast*

lemma *split_invL_sym*[*dest*]: ⟦(*l*,*b*,*r*) = *split x t*; *inv t*⟧ ⟹ *inv l*
using *split_invL* **by** *metis*

lemma *split_invR_sym*[*dest*]: ⟦(*l*,*b*,*r*) = *split x t*; *inv t*⟧ ⟹ *inv r*
using *split_invR* **by** *metis*

lemma *obtain_inv*[*elim*]:
assumes *inv* (*Node l* (*x*,*y*) *r*)
obtains *inv l inv r*
using *assms inv_Node* **by** *blast*

The decreasing side of the submodularity rule of [4] is the instance $x = 0$.

lemma *rule_sub_dec*:
⟦ *rank l*′ ≤ *rank l*; *rank r*′ ≤ *rank r*; *inv l*′; *inv r*′; *inv* (*Node l* (*a*,*b*) *r*) ⟧ ⟹
rank (*join l*′ *a r*′) ≤ *rank* (*Node l* (*a*,*b*) *r*)
using *rule_sub*[*of l*′ *l* 0 *r*′ *r a b*] **by** *simp*

Since *join* can build a tree of any size out of the empty tree, the invariant holds for a singleton node with any pivot. Applying *rule_sub* to such a node bounds the rank increase of any join by a constant [4, Property 6].

lemma *inv_singleton*: ∃ *b*. *inv* (*Node Leaf* (*a*,*b*) *Leaf*)
proof −
have *inv* (*join Leaf a Leaf*) **and** *size* (*join Leaf a Leaf*) = *1*
by (*auto simp*: *inv_Leaf inv_join join_size*)
moreover have *set_tree* (*join Leaf a Leaf*) = {*a*}
by *simp*
ultimately show *?thesis*
by (*cases join Leaf a Leaf*) *auto*
qed

lemma *join_rank_upper*:
assumes *inv l inv r*
shows *rank* (*join l a r*) ≤ *max* (*rank l*) (*rank r*) + c_u
proof −
obtain *b* **where** *inv_s*: *inv* (*Node Leaf* (*a*,*b*) *Leaf*)
using *inv_singleton* **by** *blast*
then have *rank* (*Node Leaf* (*a*,*b*) *Leaf*) ≤ c_u
using *inv_children_explD*(*3*)[*OF inv_s*] *rule_empty* **by** *simp*
moreover have *rank* (*join l a r*) ≤ *rank* (*Node Leaf* (*a*,*b*) *Leaf*) + *max* (*rank l*) (*rank r*)

using *rule_sub rule_empty assms inv_s* **by** *force*
ultimately show *?thesis*
by *argo*
qed

4.1 Work model for *join*

All following work analyses are carried out against the idealized work measures in which every *join* is charged exactly *W_join l a r*. Through *rule_cost* every generated time function is bounded by a constant multiple of its work-model counterpart. It would be possible to use *T_join* directly, but the actual bounds will be calculated via real analysis, rather than just summation in *nat*. This way the constants do not need to be dragged around and conversions between *nat* and *real* happen exactly once.

definition *W_join* :: $('a * 'b)$ *tree* $\Rightarrow$ $'a \Rightarrow$ $('a * 'b)$ *tree* $\Rightarrow$ *real* **where**
W_join l a r = $|$*rank l* $-$ *rank r*$|$

definition *K_T* :: *real* **where**
K_T = *1* + *2* $*$ k_0 + k_1

lemma *K_T_ge_1*: *1* $\leq$ *K_T*
using k_0*_nonneg* k_1*_nonneg* **by** (*simp add*: *K_T_def*)

lemma *K_T_ge_k1*: k_1 $\leq$ *K_T*
using k_0*_nonneg* **by** (*simp add*: *K_T_def*)

lemma *K_T_ge_1_plus_k0*: *1* + k_0 $\leq$ *K_T*
using k_0*_nonneg* k_1*_nonneg* **by** (*simp add*: *K_T_def*)

lemma *W_join_nonneg*[*simp*]: *0* $\leq$ *W_join l a r*
by (*simp add*: *W_join_def*)

lemma *rule_cost_W*:
$[\![$*inv l*; *inv r*$]\!] \Longrightarrow$ *real* (*T_join l a r*) $\leq$ k_0 + k_1 $*$ *W_join l a r*
by (*simp add*: *W_join_def rule_cost*)

4.2 Work bounds of helper functions

4.2.1 *split*

First, show that *split* can be decomposed into two functions, computing the left and right subtree respectively.

fun *splitl* :: $('a * 'b)$ *tree* $\Rightarrow$ $'a \Rightarrow$ $('a * 'b)$ *tree* **where**
splitl Leaf x = *Leaf* |
splitl (*Node l* (*a*,_) *r*) *x* = (*case* (*cmp x a*) *of*
LT $\Rightarrow$ *splitl l x* |
EQ $\Rightarrow$ *l* |
GT $\Rightarrow$ *join l a* (*splitl r x*))

```
fun splitr :: ('a * 'b) tree ⇒ 'a ⇒ ('a * 'b) tree where
splitr Leaf x = Leaf |
splitr (Node l (a, _) r) x = (case (cmp x a) of
   LT ⇒ join (splitr l x) a r |
   EQ ⇒ r |
   GT ⇒ splitr r x)

lemma splitl_eq_l:split x t = (l,b,r) ⟹ splitl t x = l
  by(induction t arbitrary: l b r)(auto simp: split.simps split!: prod.splits)

lemma splitr_eq_r:split x t = (l,b,r) ⟹ splitr t x = r
  by(induction t arbitrary: l b r)(auto simp: split.simps split!: prod.splits)

corollary splitl_fst: splitl t x = fst (split x t)
  using splitl_eq_l by (cases split x t) force

corollary splitr_snd: splitr t x = snd (snd (split x t))
  using splitr_eq_r by (cases split x t) force

corollary inv_splitl:inv t ⟹ inv (splitl t x)
  by (metis split_inv splitl_eq_l prod_cases3)

corollary inv_splitr:inv t ⟹ inv (splitr t x)
  by (metis split_inv splitr_eq_r prod_cases3)
```

The submodularity rule guarantees that the rank of either result trees can never exceed the rank of the original input tree.

```
lemma splitl_rank:inv t ⟹ rank t ≥ rank (splitl t x)
proof(induction t rule: tree2_induct)
  case Leaf
  then show ?case
    by simp
next
  case (Node l a b r)
  then have t_bal:balanced (Node l (a,b) r)
    using rule_bal by fast

  then show ?case
  proof(cases cmp x a)
    case GT
    have rank r ≥ rank (splitl r x)
      using Node inv_Node by blast
    moreover have rank l ≥ rank l
      by simp
    ultimately have rank (Node l (a,b) r) ≥ rank (join l a (splitl r x))
      by (meson Node.prems inv_Node rule_sub_dec inv_splitl)
    then show ?thesis
      using GT by simp
```

```
  qed (use Node t_bal c_vals in auto)
qed

lemma splitr_rank:inv t ⟹ rank t ≥ rank (splitr t x)
proof(induction t rule: tree2_induct)
  case Leaf
  then show ?case
    by simp
next
  case (Node l a b r)
  then have t_bal:balanced (Node l (a,b) r)
    using rule_bal by fast
  then show ?case
  proof(cases cmp x a)
    case LT
    have rank l ≥ rank (splitr l x)
      using Node inv_Node by blast
    moreover have rank r ≥ rank r
      by simp
    ultimately have rank (Node l (a,b) r) ≥ rank (join (splitr l x) a r)
      by (meson Node.prems inv_Node rule_sub_dec inv_splitr)
    then show ?thesis
      using LT by simp
  qed (use Node t_bal c_vals in auto)
qed

corollary split_rank: ⟦split x t = (l,b,r); inv t⟧ ⟹ rank l ≤ rank t ∧ rank r ≤
rank t
  using splitl_eq_l splitl_rank splitr_eq_r splitr_rank by blast
```

At most one of the two results of a split can retain the full rank of the input. Splitting at the root returns the two subtrees, and otherwise the recursion descends into one subtree, whose share of the result is bounded by that subtree's rank.

```
lemma split_rank_min:
  assumes inv t t ≠ Leaf split x t = (l, b, r)
  shows min (rank l) (rank r) ≤ rank t − c_l
proof −
  obtain tl a c tr where t: t = Node tl (a,c) tr
    using assms(2) by (cases t rule: tree2_cases) auto
  moreover have inv_sub: inv tl inv tr
    using assms(1) t inv_Node by blast+
  moreover have bl: rank tl ≤ rank t − c_l rank tr ≤ rank t − c_l
    using inv_children_explD assms t by auto
  moreover have l_eq: l = splitl t x and r_eq: r = splitr t x
    using assms(3) splitl_eq_l splitr_eq_r by auto
  ultimately show ?thesis
  by (cases cmp x a)
        (use splitl_rank splitr_rank splitl.simps in auto; metis min.absorb1
```

min_le_iff_disj)+
qed

Furthermore, since *split* can only partition keys, the combined sizes of the results never exceed the size of the input.

lemma *split_size_sum*:
⟦(*l*, *b*, *r*) = *split x t*; *inv t*⟧ ⟹
inv l ∧ *inv r* ∧ *size t* = *size l* + *size r* + (*if b then 1 else 0*)
by (*induction t arbitrary*: *l b r rule*: *split.induct*)
(*auto simp*: *split.simps join_size inv_join inv_Leaf dest*!: *inv_Node*
split: *cmp_val.splits prod.splits if_splits*)

corollary *split_size*: ⟦(*l*, *b*, *r*) = *split x t*; *inv t*⟧ ⟹ *size l* + *size r* ≤ *size t*
using *split_size_sum* **by** *simp*

And exactly one element is removed if the split key was contained in the original tree.

lemma *split_size_true*:
⟦(*l*, *True*, *r*) = *split x t*; *inv t*⟧ ⟹ *size l* + *size r* = *size t* − *1*
using *split_size_sum* **by** *simp*

The work of *split* is also partitioned by the work of computing the left/right result.

fun *W_split* :: ('*a* ∗ '*b*) *tree* ⇒ '*a* ⇒ *real* **where**
W_split Leaf x = *1* |
W_split (*Node l* (*a*,_) *r*) *x* = (*case* (*cmp x a*) *of*
LT ⇒ *let* (*l1*,_,*l2*) = *split x l in 1* + *W_join l2 a r* + *W_split l x* |
EQ ⇒ *1* |
GT ⇒ *let* (*r1*,_,*r2*) = *split x r in 1* + *W_join l a r1* + *W_split r x*)

fun *W_splitl* :: ('*a* ∗ '*b*) *tree* ⇒ '*a* ⇒ *real* **where**
W_splitl Leaf x = *1* |
W_splitl (*Node l* (*a*,_) *r*) *x* = (*case* (*cmp x a*) *of*
LT ⇒ *W_splitl l x* |
EQ ⇒ *1* |
GT ⇒ *1* + *W_join l a* (*splitl r x*) + *W_splitl r x*)

fun *W_splitr* :: ('*a* ∗ '*b*) *tree* ⇒ '*a* ⇒ *real* **where**
W_splitr Leaf x = *1* |
W_splitr (*Node l* (*a*,_) *r*) *x* = (*case* (*cmp x a*) *of*
LT ⇒ *1* + *W_join* (*splitr l x*) *a r* + *W_splitr l x* |
EQ ⇒ *1* |
GT ⇒ *W_splitr r x*)

lemma *W_split_lr*:*W_split t x* ≤ *W_splitl t x* + *W_splitr t x*
by (*induction t rule*: *tree2_induct*)
(*auto simp*: *splitl_fst splitr_snd split*: *cmp_val.splits prod.splits*)

The bound on *W_splitl* is proved directly by induction, without the intermediary sum used in [4]. The key observation is an inverse relationship

between the cost of a single join and the cost accumulated along the recursion path. At each recursive call, *W_splitl* computes *splitl r x* and then joins the result with *l*. Writing $r' = splitl\ r\ x$:

- If *rank* r' is small (extreme: *rank* $r' = 0$), the join *W_join l a* r' is expensive (up to *rank l*), but by the monotonicity rule no expensive joins can have occurred along the path, so the accumulated cost is low
- If *rank* r' is large (extreme: *rank* r' = *rank r*), the join cost decreases, while the accumulated path cost may be higher

This tradeoff is captured by an induction hypothesis which introduces *rank* (*splitl t x*) to absorb the credit left by cheap joins.

definition $c_1 = 2 * c_\delta + 1$

lemma *c1_pos*:$c_1 > 0$
  using c_1*_def* c_δ*_def c_vals* **by** *linarith*

lemma *W_splitl_bounded*:
assumes *inv t*
shows *W_splitl t x* $\leq$ *1* + c_1 * *height t* + *rank* (*splitl t x*)
using *assms*
proof(*induction t rule*: *tree2_induct*)
  case *Leaf*
  then show *?case*
    by (*simp add*: *rule_empty*)
next
  case (*Node l a b r*)
  let *?t* = (*Node l* (*a*,*b*) *r*)

  have *t_bal*:*balanced ?t*
    using *rule_bal Node* **by** *fast*

  have *inv_subtrees*:*inv l* $\wedge$ *inv r*
    using *Node inv_Node* **by** *blast*
  show *?case*
  proof(*cases cmp x a*)
    case *LT*
    — follows directly from the IH
    have $*$:*height l* $\leq$ *height ?t*
      by *simp*
    have *W_splitl ?t x* $\leq$ *1* + c_1 * *height l* + *rank* (*splitl ?t x*)
      using *LT Node inv_subtrees* **by** *force*
    also have ... $\leq$ *1* + c_1 * *height ?t* + *rank* (*splitl ?t x*)
      using *c1_pos* **by** *simp*
    finally show *?thesis* **.**
  next
    case *EQ*

```
  then show ?thesis
    using c1_pos inv_subtrees rank_pos rule_bal by auto
 next
  case GT
  — since the rank of the split result is bounded
  have rank r ≥ rank (splitl r x)
    using Node splitl_rank inv_subtrees by blast
  — and the tree is balanced
  then have rank l + c_δ ≥ rank (splitl r x)
    using Node t_bal c_δ_def by force
  — there must be some δ that satisfies the equation below
  moreover obtain δ where hdiff:rank l + c_δ = rank (splitl r x) + δ
    by (metis add.commute diff_add_cancel)
  — thus the time to join is bounded
  ultimately have W_join l a (splitl r x) ≤ c_δ + δ
    using W_join_def c_δ_def c_vals by force
  — since x > a
  moreover have W_splitl ?t x ≤ 1 + W_join l a (splitl r x) + W_splitl r x
    using GT by simp
  — plugging in the bound for W_join l a (splitl r x)
  ultimately have W_splitl ?t x ≤ 1 + c_δ + δ + W_splitl r x
    by simp
  — applying the induction hypothesis
  then have W_splitl ?t x ≤ 1 + c_δ + δ + 1 + c_1 * height r + rank (splitl r x)
    using Node inv_subtrees by fastforce
  — expressing rank (splitl t x) in terms of rank l + c_δ and δ
  also have ... ≤ 1 + c_δ + δ + 1 + c_1 * height r + rank l + c_δ − δ
    using hdiff by simp
  — simplifying the expression
  also have ... ≤ 2 + 2*c_δ + c_1 * height r + rank l
    by simp
  — from the monotonicity rule
  also have ... ≤ 2 + 2*c_δ + c_1 * height r + rank (splitl ?t x)
    using inv_subtrees inv_splitl rule_mono inv_subtrees GT by fastforce
  — from the definition of height
  also have ... ≤ 2 + 2*c_δ + c_1 * height ?t − c_1 + rank (splitl ?t x)
    using c1_pos by (simp add: distrib_left)
  — finally, because c_1 = 2 * c_δ + 1
  also have ... ≤ 1 + c_1 * height ?t + rank (splitl ?t x)
    by (simp add: c_1_def)
  finally show ?thesis .
 qed
qed
```

The right side is mirrored.

```
lemma W_splitr_bounded:
assumes inv t
shows W_splitr t x ≤ 1 + c_1 * height t + rank (splitr t x)
using assms
```

```
proof(induction t rule: tree2_induct)
  case Leaf
  then show ?case
    by (simp add: rule_empty)
next
  case (Node l a b r)
  let ?t = (Node l (a,b) r)

  have t_bal:balanced ?t
    using rule_bal Node by fast

  have inv_subtrees:inv l ∧ inv r
    using Node inv_Node by blast
  show ?case
  proof(cases cmp x a)
    case LT
    have rank l ≥ rank (splitr l x)
      using Node splitr_rank inv_subtrees by blast
    then have rank r + c_δ ≥ rank (splitr l x)
      using Node t_bal c_δ_def by force
    then obtain δ where hdiff:rank r + c_δ = rank (splitr l x) + δ
      by (metis add.commute diff_add_cancel)
    then have W_join (splitr l x) a r ≤ c_δ + δ
     using W_join_def hdiff using ‹rank (splitr l x) ≤ rank r + c_δ› c_δ_def c_vals
by force
   then have W_splitr ?t x ≤ 1 + c_δ + δ + 1 + c_1 * height l + rank (splitr l x)
      using Node LT inv_subtrees by fastforce
    also have ... ≤ 1 + c_δ + δ + 1 + c_1 * height l + rank r + c_δ − δ
      using hdiff by simp
    also have ... ≤ 2 + 2*c_δ  + c_1 * height l + rank r
      by simp
    also have ... ≤ 2 + 2*c_δ + c_1 * height l + rank (splitr ?t x)
      using inv_subtrees inv_splitr rule_mono inv_subtrees LT by fastforce
    also have ... ≤ 2 + 2*c_δ + c_1 * height ?t − c_1 + rank (splitr ?t x)
      using c1_pos by (simp add: distrib_left)
    finally show ?thesis
      by (simp add: c_1_def)
  next
    case EQ
    then show ?thesis
      using c1_pos inv_subtrees rank_pos rule_bal by auto
  next
    case GT
    have *:height r ≤ height ?t
      by simp
    have W_splitr ?t x ≤ 1 + c_1 * height r + rank (splitr ?t x)
      using GT Node inv_subtrees by auto
    also have ... ≤ 1 + c_1 * height ?t + rank (splitr ?t x)
      by (simp add: c1_pos)
```

finally show *?thesis* **.**
qed
qed

Combining both yields the work bound of *split*.

definition *K_split* **where**
$K\_split = (4 * c_u - 2 * c_l + 2) / c_l$

lemma *K_split_pos*: $K\_split > 0$
unfolding *K_split_def* **using** *c_vals* **by** *fastforce*

lemma *K_split_ge_1*: $K\_split \geq 1$
unfolding *K_split_def* **using** *c_vals* **by** *fastforce*

theorem *W_split_bounded*:
assumes *inv t*
shows $W\_split\ t\ x \leq 2 + K\_split * rank\ t$
proof −
have $c_1 * height\ t \leq c_1 * (rank\ t\ /\ c_l)$
using *rank_lower_height_inverse'*[*OF assms*] *c1_pos*
by (*intro mult_left_mono*) *auto*
then have $W\_split\ t\ x \leq 2 + 2 * (c_1 * (rank\ t\ /\ c_l)) + 2 * rank\ t$
using *W_split_lr*[*of t x*] *W_splitl_bounded*[*OF assms*, *of x*] *W_splitr_bounded*[*OF assms*, *of x*]
splitl_rank[*OF assms*, *of x*] *splitr_rank*[*OF assms*, *of x*] **by** *linarith*
also have $\ldots = 2 + K\_split * rank\ t$
unfolding *K_split_def* c_1*_def* c_δ*_def* **using** *c_vals* **by** (*simp add*: *field_simps*)
finally show *?thesis* **.**
qed

Finally the generated *T_split* is bounded by its work-function equivalent.

time_fun *cmp*

time_fun *split* **equations** *split.simps*

lemma *W_split_ge_1*: $1 \leq W\_split\ t\ x$
by (*induction t rule*: *tree2_induct*)
(*auto simp*: *W_join_def split*: *cmp_val.splits prod.splits*)

lemma *rule_cost_step*:
assumes *inv l inv r* **and** $real\ t \leq (1 + k_0 + k_1) * w$
shows $1 + (real\ t + real\ (T\_join\ l\ a\ r)) \leq (1 + k_0 + k_1) * (1 + W\_join\ l\ a\ r + w)$
proof −
have $1 + (real\ t + real\ (T\_join\ l\ a\ r)) \leq (1 + k_0) + k_1 * W\_join\ l\ a\ r + (1 + k_0 + k_1) * w$
using *rule_cost_W*[*OF assms*(*1*,*2*), **where** *a*=*a*] *assms*(*3*) **by** *linarith*
also have $\ldots \leq (1 + k_0 + k_1) + (1 + k_0 + k_1) * W\_join\ l\ a\ r + (1 + k_0 + k_1) * w$

```
  using k0_nonneg k1_nonneg by (simp add: add_increasing mult_right_mono)
 also have ... = (1 + k0 + k1) * (1 + W_join l a r + w)
  by algebra
 finally show ?thesis .
qed

lemma T_split_bridge_strong:
 inv t ⟹ real (T_split x t) ≤ (1 + k0 + k1) * W_split t x
 using k0_nonneg k1_nonneg
 by(induction x t rule: T_split.induct)
  (auto simp: split_inv intro!: rule_cost_step split: cmp_val.splits prod.splits)

corollary T_split_bridge:
 assumes inv t
 shows real (T_split x t) ≤ K_T * W_split t x
proof −
 have (1 + k0 + k1) * W_split t x ≤ K_T * W_split t x
  using W_split_ge_1[of t x] k0_nonneg unfolding K_T_def by auto
 then show ?thesis
  using T_split_bridge_strong[OF assms] by (metis max.absorb2 max.bounded_iff)
qed
```

4.2.2 *split_min*

split_min removes the minimum element from a tree. The rank of the resulting tree stays close to the original i.e. it can decrease but not drop by more than c_u.

```
lemma split_min_rank:
 assumes inv t split_min t = (m, t') t ≠ Leaf
 shows rank t ≥ rank t'
using assms proof(induction t arbitrary: t' rule: tree2_induct)
 case Leaf then show ?case
  by auto
next
 case (Node l a b r)
 let ?t = Node l (a,b) r
 have inv_sub: inv l inv r
  using Node inv_Node by blast+
 show ?case
 proof (cases l = Leaf)
  case True
  then show ?thesis
    using Node.prems(1,2) split_min.simps c_vals inv_children_explD(2) by
force
 next
  case False
  then obtain m' l' where SmL: split_min l = (m', l')
                and inv_l': inv l'
    by (metis split_min_inv False inv_sub(1) old.prod.exhaust)
```

```
    then have rank l ≥ rank l'
      using Node False inv_sub by auto
    moreover have t' = join l' a r
      using Node False SmL by auto
    ultimately show ?thesis
      by (simp add: Node.prems(1) inv_l' inv_sub(2) rule_sub_dec)
  qed
qed

lemma split_min_rank_min:
  assumes inv t split_min t = (m, t') t ≠ Leaf
  shows rank t ≤ rank t' + c_u
using assms proof(induction t arbitrary: t' rule: tree2_induct)
  case Leaf then show ?case by auto
next
  case (Node l a b r)
  let ?t = Node l (a,b) r
  have inv_sub: inv l inv r
    using Node inv_Node by blast+
  show ?case
  proof (cases l = Leaf)
    case True
    then show ?thesis
      using Node.prems(1,2) inv_children_explD(4) by force
  next
    case False
    then obtain m' l' where SmL: split_min l = (m', l')
                        and inv_l': inv l'
      by (metis split_min_inv inv_sub(1) old.prod.exhaust)
    have rank ?t ≤ rank r + c_u
      using Node.prems(1) inv_children_explD(4) by force
    also have ... ≤ rank (join l' a r) + c_u
      using rule_mono[OF inv_l' inv_sub(2)] by auto
    moreover have t' = join l' a r
      using Node False SmL by auto
    ultimately show ?thesis by simp
  qed
qed
```

As per functional correctness, *split_min* always removes exactly one element.

```
lemma split_min_sum:
  ⟦split_min t = (m, t'); inv t; t ≠ Leaf⟧ ⟹ inv t' ∧ size t = size t' + 1
  by (induction t arbitrary: m t' rule: tree2_induct)
     (auto simp: join_size inv_join inv_Leaf dest!: inv_Node split: prod.splits
if_splits)

corollary split_min_size:
  assumes inv t split_min t = (m, t') t ≠ Leaf
```

shows *size* $t' = size\ t - 1$
using *split_min_sum assms* **by** *simp*

The work of *split_min* follows the same inductive pattern as *W_splitl*.

fun *W_split_min* :: $('a*'b)$ *tree* ⇒ *real* **where**
W_split_min (*Node l* (*a*, _) *r*) =
(*if l* = *Leaf then 1 else let* (*m*,*l'*) = *split_min l in 1* + *W_split_min l* + *W_join l' a r*)

lemma *W_split_min_bounded*:
assumes *inv t split_min t* = (*m*, *t'*) *t* ≠ *Leaf*
shows *W_split_min t* ≤ *1* + c_1 ∗ *height t* + *rank t'*
using *assms* **proof** (*induction t arbitrary*: *t' rule*: *tree2_induct*)
case *Leaf* **then show** *?case* **by** *simp*
next
case (*Node l a x r*)
let *?t* = *Node l* (*a*,*x*) *r*
have *inv_sub*: *inv l* ∧ *inv r* **using** *inv_Node Node* **by** *blast*
have *bal*: *balanced ?t* **using** *Node* **by** *blast*
show *?case*
proof (*cases l* = *Leaf*)
case *True*
then have *t'* = *r* **and** *W_split_min ?t* = *1*
using *Node* **by** *auto*
then show *?thesis* **using** c_1*_def rank_pos c1_pos inv_sub* **by** *force*
next
case *False*
obtain *m' l'* **where** *SmL*: *split_min l* = (*m'*, *l'*)
using *False* **by** *fastforce*
have *IH*: *W_split_min l* ≤ *1* + c_1 ∗ *height l* + *rank l'*
using *Node False inv_sub SmL* **by** *simp*
have *t'_def*: *t'* = *join l' a r*
using *Node False SmL* **by** *auto*
have *unfold*: *W_split_min ?t* = *1* + *W_split_min l* + *W_join l' a r*
using *False SmL* **by** *force*
have *rm*: *rank l* ≥ *rank l'*
using *split_min_rank False inv_sub SmL* **by** *simp*
then have *rank r* + c_δ ≥ *rank l'*
using *Node bal* c_δ*_def* **by** *force*
then obtain δ **where** *hdiff*: *rank r* + c_δ = *rank l'* + δ
by (*metis add.commute diff_add_cancel*)
then have *W_join l' a r* ≤ c_δ + δ
using *W_join_def* c_δ*_def c_vals* ‹*rank l'* ≤ *rank r* + c_δ› **by** *force*
then have *W_split_min ?t* ≤ *2* + c_1 ∗ *height l* + *rank l'* + c_δ + δ
using *IH unfold* **by** *linarith*
also have ... = *2* + *2* ∗ c_δ + c_1 ∗ *height l* + *rank r*
using *hdiff* **by** *simp*
also have ... ≤ *2* + *2*∗c_δ + c_1 ∗ *height l* + *rank t'*
using *inv_sub rule_mono False SmL split_min_inv t'_def* **by** *force*

```
    also have ... ≤ 2 + 2*c_δ + c_1 * height ?t − c_1 + rank t'
      using c1_pos by (simp add: distrib_left)
    also have ... ≤ 1 + c_1 * height ?t + rank t'
      by (simp add: c_1_def)
    finally show ?thesis
      by fastforce
  qed
qed

definition K_min :: real where K_min = 1 + c_1 / c_l

lemma K_min_nonneg: 0 ≤ K_min
  using K_min_def c1_pos c_vals by auto

corollary W_split_min_rank:
  assumes inv t split_min t = (m, t') t ≠ Leaf
  shows W_split_min t ≤ 1 + K_min * rank t
proof −
  have W_split_min t ≤ 1 + c_1 * height t + rank t'
    using assms W_split_min_bounded by force
  also have ... ≤ 1 + c_1 * (rank t / c_l) + rank t'
    using c1_pos assms(1) mult_left_mono rank_lower_height_inverse by fast-
force
  also have ... ≤ 1 + c_1 * (rank t / c_l) + rank t
    by (simp add: assms split_min_rank)
  also have ... ≤ 1 + (1 + c_1 / c_l) * rank t
    by argo
  finally show ?thesis
    using K_min_def by presburger
qed

lemma W_split_min_ge_1: t ≠ Leaf ⟹ 1 ≤ W_split_min t
  by (induction t rule: tree2_induct)
     (auto simp: W_join_def split: prod.splits)
```

T_split_min is then bounded by *W_split_min*.

```
time_fun split_min equations split_min.simps

lemma rule_cost_step_min:
  assumes inv l inv r and real t ≤ (1 + k_0 + k_1) * w
  shows 1 + (real t + real (T_join l a r)) ≤ (1 + k_0 + k_1) * (1 + w + W_join
l a r)
  using rule_cost_step[OF assms] by (simp add: ac_simps)

lemma T_split_min_bridge_strong:
  ⟦inv t; t ≠ Leaf⟧ ⟹ real (T_split_min t) ≤ (1 + k_0 + k_1) * W_split_min t
  using k_0_nonneg k_1_nonneg
  by (induction t rule: tree2_induct)
       (auto intro!: rule_cost_step_min dest: split_min_inv inv_Node split:
```

prod.splits)

4.2.3 *join2*

join2 joins two trees without a pivot element by extracting the minimum of the right tree. Its rank is close to the maximum of its inputs, analogous to the monotonicity rule.

lemma *join2_split_min*: $[\![split\_min\ r = (m,\ r');\ r \neq Leaf]\!] \Longrightarrow join2\ l\ r = join\ l\ m\ r'$
  by (*simp add*: *join2_def*)

lemma *rule_mono_join2*:
  assumes *inv l inv r*
  shows *max* (*rank l*) (*rank r*) $\leq$ *rank* (*join2 l r*) + c_u
proof (*cases r*)
  case *Leaf*
  then show *?thesis*
    using *c_vals join2_def assms*(*1*) *rank_pos rule_bal rule_empty* **by** *simp*
next
  case (*Node l2 x r2*)
  obtain *m r'* **where** *SmR*: *split_min r* = (*m*, *r'*)
    by (*cases split_min r*) *auto*
  have *inv_r'*: *inv r'*
    using *split_min_inv*[*OF SmR assms*(*2*)] *Node* **by** *simp*
  have *join2 l r* = *join l m r'*
    using *SmR Node* **by** (*simp add*: *join2_split_min*)
  moreover have *max* (*rank l*) (*rank r'*) $\leq$ *rank* (*join l m r'*)
    using *rule_mono*[*OF assms*(*1*) *inv_r'*] **.**
  moreover have *rank r* $\leq$ *rank r'* + c_u
    using *split_min_rank_min*[*OF assms*(*2*) *SmR*] *Node* **by** *auto*
  ultimately have *rank l* $\leq$ *rank* (*join2 l r*) + c_u
          and *rank r* $\leq$ *rank* (*join2 l r*) + c_u
    using *c_vals* **by** *auto*
  then show *?thesis*
    by (*simp add*: *max_def*)
qed

Size is again defined by functional correctness.

lemma *join2_size*:
  assumes *inv l inv r*
  shows *size* (*join2 l r*) = *size l* + *size r*
proof (*cases r*)
  case *Leaf*
  then show *?thesis* **by** (*simp add*: *join2_def*)
next
  case (*Node l2 x r2*)
  obtain *m r'* **where** *SM*: *split_min r* = (*m*, *r'*)
    by (*cases split_min r*) *auto*
  have *inv_r'*: *inv r'*

```
    using split_min_inv[OF SM assms(2)] Node by simp
  have size r' = size r - 1
    using split_min_size[OF assms(2) SM] Node by simp
  moreover have join2 l r = join l m r'
    using SM Node by (simp add: join2_split_min)
  moreover have 0 < size r
    using Node by simp
  ultimately show ?thesis
    by (simp add: join_size[OF assms(1) inv_r'])
qed
```

The work of *join2* is bounded by a join plus a linear term in the rank of the right tree, accounting for the cost of extracting the minimum.

```
definition W_join2 :: ('a*'b) tree ⇒ ('a*'b) tree ⇒ real where
  W_join2 l r = (if r = Leaf then 1
                 else let (m, r') = split_min r in W_join l m r' + W_split_min r)

lemma W_join2_split_min:
  ⟦split_min r = (m, r'); r ≠ Leaf⟧ ⟹
     W_join2 l r = W_join l m r' + W_split_min r
  by (simp add: W_join2_def)

lemma W_join2_bounded:
  assumes inv l inv r
  shows W_join2 l r ≤ W_join l a r + 1 + c_u + K_min * rank r
proof (cases r)
  case Leaf
  then show ?thesis
    using W_join2_def W_join_def c_vals rule_empty by auto
next
  case (Node l2 x r2)
  obtain m r' where SM: split_min r = (m, r')
    by (cases split_min r) auto
  have inv_r': inv r' and rank_r': rank r' ≤ rank r and rank_min: rank r ≤
rank r' + c_u
    using split_min_inv[OF SM assms(2)] split_min_rank[OF assms(2) SM]
      split_min_rank_min[OF assms(2) SM] Node by auto
  have W_join2 l r = W_join l m r' + W_split_min r
    using SM Node by (simp add: W_join2_split_min)
  also have ... ≤ W_join l m r' + 1 + K_min * rank r
    using W_split_min_rank Node assms(2) SM by force
  also have ... ≤ abs(rank l - rank r') + 1 + K_min * rank r
    using W_join_def by force
  also have ... ≤ abs(rank l - rank r) + c_u + 1 + K_min * rank r
    using rank_min rank_r' by argo
  finally show ?thesis
    using Node W_join_def by auto
qed
```

T_join2 is again bounded by *W_join2*.

time_fun *join2* **equations** *join2_def*

lemma *rule_cost_step_join2*:
assumes *inv l inv r* **and** *real t* $\leq$ (*1* + k_0 + k_1) $*$ *w* **and** *1* $\leq$ *w*
shows *real t* + *real* (*T_join l a r*) $\leq$ *K_T* $*$ (*W_join l a r* + *w*)
proof −
have $k_0 * 1 \leq k_0 * w$
using *assms*(*4*) k_0*_nonneg* **by** (*intro mult_left_mono*) *auto*
then have *absorb*: k_0 + (*1* + k_0 + k_1) $*$ *w* $\leq$ *K_T* $*$ *w*
unfolding *K_T_def* **by** *argo*

have *real t* + *real* (*T_join l a r*)
$\leq$ (k_0 + (*1* + k_0 + k_1) $*$ *w*) + k_1 $*$ *W_join l a r*
using *rule_cost_W*[*OF assms*(*1*,*2*), **where** *a*=*a*] *assms*(*3*) **by** *linarith*
also have ... $\leq$ *K_T* $*$ *w* + *K_T* $*$ *W_join l a r*
using *absorb K_T_ge_k1* **by** (*auto intro*!: *add_mono mult_right_mono*)
also have ... = *K_T* $*$ (*W_join l a r* + *w*)
by *algebra*
finally show *?thesis* **.**
qed

lemma *T_join2_bridge*:
assumes *inv l inv r*
shows *real* (*T_join2 l r*) $\leq$ *K_T* $*$ *W_join2 l r*
proof (*cases r* = *Leaf*)
case *True*
then show *?thesis* **using** *K_T_ge_1* **by** (*simp add*: *W_join2_def*)
next
case *False*
obtain *m r′* **where** *SM*: *split_min r* = (*m*,*r′*) **by** *fastforce*
have *inv_r′*: *inv r′* **using** *split_min_inv*[*OF SM assms*(*2*) *False*] **.**
show *?thesis*
using *rule_cost_step_join2*[*OF assms*(*1*) *inv_r′*
T_split_min_bridge_strong[*OF assms*(*2*) *False*]
W_split_min_ge_1[*OF False*]]
by (*simp add*: *W_join2_def False SM*)
qed

5 Layered analysis of split-work

The set operations *union*, *inter* and *diff* share one recursion pattern. At each step the root key of one tree acts as a pivot at which the other tree is split. Following [4], the tree supplying the pivots is called the pivot tree and the tree being split the decomposed tree. In *Set2_Join*, union and intersection split *t2* by the keys of *t1*, so the pivot tree is *t1* and the decomposed tree is *t2*; for difference the roles are swapped. A subtree resulting from a split is referred to as a part of the decomposed tree. *split_work* records exactly the split cost incurred along this shared recursion. The goal of this section

is obtaining a closed-form bound on *split_work*, and this bound is in essence already the work bound of the set operations. By *W_split_bounded* each individual split costs linearly in the rank of the part t it splits, i.e. logarithmically in its size. What the analysis will show is that paying this logarithmic charge once per part already sums to the optimal $O(m \log(n/m + 1))$. This will be done by grouping the parts into unit-rank layers according to the pivot tree that produced them and using the fact that the number of rank roots per layer decays geometrically (c.f. *rank_roots_count_powr_le*), making the total collapse into a geometric series. This corresponds to the exact analytical chain in [4, Theorem 11]. For the final work bounds of each set function it will be shown that the work performed by *join* along the way follows the same cost model.

```
fun split_work :: ('a * 'b) tree ⇒ ('a * 'b) tree ⇒ real where
split_work t1 t2 =
  (if t1 = Leaf then 0
   else if t2 = Leaf then 0
   else case t1 of Node l1 (a,_) r1 ⇒
     (let (l2, _, r2) = split a t2
      in W_split t2 a
         + split_work l1 l2
         + split_work r1 r2))

declare split_work.simps[simp del]
```

5.1 Layered decomposition of *split_work*

To enable layered analysis, this section shows that *split_work* can be rewritten as a double sum over all layers, summing work of individual layers. First *split_parts* is defined, which collects all parts of the decomposed tree along the recursive path.

```
fun split_parts :: (('a::linorder) * 'b) tree ⇒ ('a * 'b) tree ⇒ ('a * 'b) tree list
where
split_parts t1 t2 =
  (if t1 = Leaf then []
   else if t2 = Leaf then []
   else case t1 of
     Node l1 (a,_) r1 ⇒
       let (l2, _, r2) = split a t2
       in t2 # (split_parts l1 l2 @ split_parts r1 r2))

declare split_parts.simps [simp del]
```

Which enables bounding *split_work* by a direct sum, which already plugs in bounds for *W_split*.

```
lemma split_work_le_part_sum:
  assumes inv t2
  shows split_work t1 t2 ≤ (∑ t ←(split_parts t1 t2). 2 + K_split * rank t)
```

using *assms*
proof (*induction t1 t2 rule*: *split_work.induct*)
case (*1 t1 t2*)
then show *?case*
by (*fastforce simp*: *split_work.simps*[*of t1 t2*] *split_parts.simps*[*of t1 t2*]
split_inv add.assoc[*symmetric*]
intro!: *add_mono W_split_bounded*
split: *tree.splits prod.splits*)
qed

All parts conform to *inv*, since split preserves it.

lemma *split_parts_inv*:
assumes *inv t2*
shows $t \in$ *set* (*split_parts t1 t2*) $\Longrightarrow$ *inv t*
using *assms*
proof (*induction t1 t2 rule*: *split_parts.induct*)
case (*1 t1 t2*)
then show *?case*
by (*fastforce simp*: *split_parts.simps*[*of t1 t2*] *split*: *tree.splits prod.splits if_splits*)
qed

Next, *parts_in_layer* collects the parts belonging to specific layers of the pivot tree *t1*.

fun *parts_in_layer* :: (('*a*::*linorder*) $*$ '*b*) *tree* $\Rightarrow$ ('*a* $*$ '*b*) *tree* $\Rightarrow$ *nat* $\Rightarrow$ ('*a* $*$ '*b*) *tree list* **where**
parts_in_layer t1 t2 i =
(*if t1* = *Leaf then* []
else if t2 = *Leaf then* []
else case t1 of
Node l1 (*a*,_) *r1* $\Rightarrow$
let (*l2*, _, *r2*) = *split a t2*
in ((*if in_band i* (*rank t1*) *then* [*t2*] *else* [])
@ *parts_in_layer l1 l2 i*
@ *parts_in_layer r1 r2 i*))

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

When the rank of the pivot tree *t1* is below layer *i*, no parts can be collected in that layer. This is because the layer condition *real i* $\leq$ *rank t1* fails at the root, and by balance, all children have even smaller rank.

lemma *parts_in_layer_empty_if_rank_lt*:
assumes *balanced t1 rank t1* $<$ *i*
shows *parts_in_layer t1 t2 i* = []
using *assms*
proof (*induction t1 arbitrary*: *t2 i*)
case *Leaf* **then show** *?case*
by (*simp add*: *parts_in_layer.simps*)
next

```
  case (Node l1 x r1)
  then have rank l1 < i rank r1 < i
    using balanced_children_explD[of l1 x r1] c_vals by linarith+
  then show ?case using Node
    by (auto simp: parts_in_layer.simps[of Node l1 x r1 t2 i]
          split: prod.splits tree.splits)
qed
```

Layers above the rank of *t1* contribute nothing, so extending the summation range is harmless.

```
lemma sum_parts_in_layer_pad:
  assumes balanced t1 nat ⌈rank t1⌉ ≤ N
  shows (∑ i ≤ N. ∑ t←parts_in_layer t1 t2 i. f t)
      = (∑ i ≤ nat ⌈rank t1⌉. ∑ t←parts_in_layer t1 t2 i. f t)
proof −
  have (∑ t←parts_in_layer t1 t2 i. f t) = 0
    if i ∈ {..N} − {..nat ⌈rank t1⌉} for i
  proof −
    have nat ⌈rank t1⌉ < i
      using that by simp
    then have rank t1 < real i
      by linarith
    then show ?thesis
      by (auto simp add: parts_in_layer_empty_if_rank_lt[OF assms(1)])
  qed
  then show ?thesis
    using assms(2) by (auto intro!: sum.mono_neutral_right)
qed
```

A band condition $real\ j \le x \wedge x < real\ j + 1$ holds for exactly one index j (i.e. the floor of x). Summing an indicator over any range covering it therefore collects its value exactly once.

```
lemma sum_band_single:
  fixes x :: real
  assumes 0 ≤ x and nat ⌊x⌋ ≤ R
  shows (∑ j ≤ R. (if in_band j x then c else 0)) = c
proof −
  have (real j ≤ x ∧ x < real j + 1) ⟷ ⌊x⌋ = int j for j
    by linarith
  then have (real j ≤ x ∧ x < real j + 1) ⟷ j = nat ⌊x⌋ for j
    using assms(1) by fastforce
  then show ?thesis
    using assms(2) by force
qed
```

This now begets the central layering identity. Summing any weight over parts layer by layer yields the same value as summing over *split_parts* directly.

```
lemma sum_parts_as_layers:
```

fixes $f :: ('a * 'b)\ tree \Rightarrow real$
assumes *balanced t1*
shows $(\sum t{\leftarrow}split\_parts\ t1\ t2.\ f\ t) = (\sum i \leq nat\ \lceil rank\ t1 \rceil.\ \sum t{\leftarrow}parts\_in\_layer\ t1\ t2\ i.\ f\ t)$
using *assms*
proof (*induction t1 arbitrary*: *t2 rule*: *tree2_induct*)
case *Leaf*
then show *?case*
by (*simp add*: *split_parts.simps parts_in_layer.simps*)
next
case (*Node l1 a bx r1*)
obtain *l2 b r2* **where** *SPL*: *split a t2* = (*l2*, *b*, *r2*)
by (*cases split a t2*) *auto*

let *?t1* = *Node l1* (*a*, *bx*) *r1*
let *?r* = *nat* $\lceil$*rank ?t1*$\rceil$

have *Bl*: *balanced l1* **and** *Br*: *balanced r1*
using *Node.prems* **by** *auto*

have *rank l1* $\leq$ *rank ?t1 rank r1* $\leq$ *rank ?t1*
using *balanced_children_explD*[*OF Node.prems*] *c_vals* **by** *linarith*+
then have *Rl*: *nat* $\lceil$*rank l1*$\rceil$ $\leq$ *?r* **and** *Rr*: *nat* $\lceil$*rank r1*$\rceil$ $\leq$ *?r*
by *linarith*+

show *?case*
proof (*cases t2* = *Leaf*)
case *True*
then show *?thesis*
by (*simp add*: *split_parts.simps parts_in_layer.simps*)
next
case *False*

— The root contribution is collected exactly once, at layer $\lfloor$*rank ?t1*$\rfloor$
have *root_sum*:
$(\sum j \leq ?r.\ (if\ real\ j \leq rank\ ?t1 \wedge rank\ ?t1 < real\ j + 1\ then\ f\ t2\ else\ 0)) = f\ t2$
using *rank_pos*[*OF Node.prems*] **by** (*auto intro*!: *sum_band_single*; *linarith*)

have $(\sum t{\leftarrow}parts\_in\_layer\ ?t1\ t2\ j.\ f\ t)$
$= (if\ real\ j \leq rank\ ?t1 \wedge rank\ ?t1 < real\ j + 1\ then\ f\ t2\ else\ 0)$
$+ ((\sum t{\leftarrow}parts\_in\_layer\ l1\ l2\ j.\ f\ t) + (\sum t{\leftarrow}parts\_in\_layer\ r1\ r2\ j.\ f\ t))$ **for** *j*
using *False SPL*
by (*auto simp*: *parts_in_layer.simps*[*of ?t1 t2 j*] *split*: *prod.splits tree.splits*)
then have $(\sum i \leq ?r.\ \sum t{\leftarrow}parts\_in\_layer\ ?t1\ t2\ i.\ f\ t)$
$= (\sum j \leq ?r.\ (if\ real\ j \leq rank\ ?t1 \wedge rank\ ?t1 < real\ j + 1\ then\ f\ t2\ else\ 0)) +$
$((\sum i \leq ?r.\ \sum t{\leftarrow}parts\_in\_layer\ l1\ l2\ i.\ f\ t) +$

```
        (∑ i ≤ ?r. ∑ t←parts_in_layer r1 r2 i. f t))
    by (simp add: sum.distrib)
  also have ... = f t2 + ((∑ t←split_parts l1 l2. f t) + (∑ t←split_parts r1
r2. f t))
  by (simp add: root_sum sum_parts_in_layer_pad[OF Bl Rl] sum_parts_in_layer_pad[OF
Br Rr]
                Node.IH(1)[OF Bl] Node.IH(2)[OF Br])
  also have ... = (∑ t←split_parts ?t1 t2. f t)
    using False SPL by (simp add: split_parts.simps)
  finally show ?thesis
    by fastforce
 qed
qed
```

5.1.1 Bounding *parts_in_layer* via rank roots

With the layering identity in place, one piece is still missing. Per layer, *parts_in_layer* accounts for the cost accurately but has no usable structure. Its entries are nested fragments of each other, so neither their number nor their total size can be nicely argued about. The coarsening *rank_root_parts_in_layer* walks down *t1* only until a node whose rank falls in the band $[i, i+1)$ (i.e. it is a rank root) records the current part and stops.

```
fun rank_root_parts_in_layer :: (('a::linorder) * 'b) tree ⇒ ('a * 'b) tree ⇒ nat
⇒ ('a * 'b) tree list where
rank_root_parts_in_layer t1 t2 i =
 (if t1 = Leaf then []
  else if t2 = Leaf then []
  else case t1 of
    Node l1 (a,_) r1 ⇒
      let (l2, _, r2) = split a t2
      in (if in_band i (rank t1) then [t2] else
          rank_root_parts_in_layer l1 l2 i
          @ rank_root_parts_in_layer r1 r2 i))

declare rank_root_parts_in_layer.simps[simp del]
```

All such rank root parts in a layer preserve the invariant.

```
lemma root_parts_inv:
 assumes inv t2
 shows t ∈ set (rank_root_parts_in_layer t1 t2 i) ⟹ inv t
 using assms
proof (induction t1 t2 i rule: rank_root_parts_in_layer.induct)
 case (1 t1 t2 i)
 then show ?case
   by (fastforce simp: rank_root_parts_in_layer.simps[of t1 t2 i] split_inv
       split: tree.splits prod.splits if_splits)
qed
```

Moreover, none of them is empty as the recursion stops as soon as the decomposed tree runs out.

lemma *root_parts_nonempty*:
t ∈ *set* (*rank_root_parts_in_layer t1 t2 i*) ⟹ *t* ≠ *Leaf*
proof (*induction t1 t2 i rule*: *rank_root_parts_in_layer.induct*)
case (*1 t1 t2 i*)
then show *?case*
by (*fastforce simp*: *rank_root_parts_in_layer.simps*[*of t1 t2 i*]
split: *tree.splits prod.splits if_splits*)
qed

The combined size of all rank root parts in a single layer cannot exceed the size of the decomposed tree *t2* [4, Lemma 10].

lemma *sizes_root_layer*:
assumes *inv t2*
shows (∑ *t* ← *rank_root_parts_in_layer t1 t2 i*. *size t*) ≤ *size t2*
using *assms*
proof (*induction t1 arbitrary*: *t2 i*)
case *Leaf* **then show** *?case*
by (*simp add*: *rank_root_parts_in_layer.simps*)
next
case (*Node l1 x r1*)
obtain *l2 b r2* **where** *SPL*: *split* (*fst x*) *t2* = (*l2*, *b*, *r2*)
by (*cases split* (*fst x*) *t2*) *auto*

have *inv l2 inv r2*
using *SPL Node.prems split_inv* **by** *blast*+
then have (∑ *t*←*rank_root_parts_in_layer l1 l2 i*. *size t*) +
(∑ *t*←*rank_root_parts_in_layer r1 r2 i*. *size t*) ≤ *size t2*
using *Node.IH*(*1*) *Node.IH*(*2*) *split_size*[*OF SPL*[*symmetric*] *Node.prems*]
by (*meson add_mono_thms_linordered_semiring*(*1*) *dual_order.trans*)
then show *?case*
using *SPL* **by** (*auto simp*: *rank_root_parts_in_layer.simps*[*of Node l1 x r1 t2 i*]
split: *tree.splits prod.splits if_splits*)
qed

The number of rank root parts per layer is bounded directly by the number of rank roots of the pivot tree.

lemma *length_root_parts_le_rank_roots*:
length (*rank_root_parts_in_layer t1 t2 i*) ≤ *length* (*rank_roots t1 i*)
by (*induction t1 arbitrary*: *t2*)
(*auto simp*: *rank_root_parts_in_layer.simps add_mono split*: *prod.splits tree.splits if_splits*)

Each part collected by *parts_in_layer* is bounded by the rank of the decomposed tree.

lemma *part_rank_le_snd*:

```
  t ∈ set (parts_in_layer t1 t2 i) ⟹ inv t2 ⟹ rank t ≤ rank t2
proof (induction t1 arbitrary: t2)
  case Leaf
  then show ?case
    by (simp add: parts_in_layer.simps)
next
  case (Node l x r)
  show ?case
  proof (cases t2 = Leaf)
    case True
    then show ?thesis
      using Node.prems by (simp add: parts_in_layer.simps)
  next
    case False
    obtain l2 b r2 where SPL: split (fst x) t2 = (l2, b, r2)
      by (cases split (fst x) t2) auto
    have inv l2 and inv_r2: inv r2
      using SPL Node.prems(2) split_inv by metis+
    moreover have rank l2 ≤ rank t2 rank r2 ≤ rank t2
      using SPL Node.prems(2) split_rank by blast+
    ultimately show ?thesis
      using Node SPL False
      by (fastforce simp: parts_in_layer.simps[of Node l x r t2]
                    intro: order.trans
                    split: prod.splits tree.splits if_splits)
  qed
qed
```

Therefore, at a rank root *t1* of layer *i*, the number of parts in *parts_in_layer t1 t2 i* is bounded by the constant C i.e. the layer of *t1* has at most C nodes (c.f. *layer_bound_const*).

```
lemma length_parts_in_layer_le_C:
  assumes balanced t1 in_band i (rank t1)
  shows length (parts_in_layer t1 t2 i) ≤ C
proof −
  have length (parts_in_layer t1 t2 i) ≤ length (layer t1 i)
  proof (induction t1 arbitrary: t2)
    case Leaf then show ?case
      by (simp add: parts_in_layer.simps)
  next
    case (Node l ab r)
    then show ?case
      by (auto simp: parts_in_layer.simps[of Node l ab r t2]
                     add_mono_thms_linordered_semiring(1)
               split: prod.splits tree.split if_split)
  qed
  also have ... ≤ C
    using layer_bound_const assms by (simp add: C_def)
  finally show ?thesis .
```

qed

Consequently, at a rank root the whole weighted part sum is charged against a single part. There are at most C parts, each of rank at most *rank t2*.

lemma *sum_parts_band_le_C*:
assumes *balanced t1 inv t2 in_band i* (*rank t1*)
and $k > 0$ $c > 0$
shows $(\sum t \leftarrow$ *parts_in_layer t1 t2 i*. $k + c *$ *rank t*$) \leq$ *real C* $* (k + c *$ *rank t2*$)$
proof −
have *each*: $\bigwedge$*t*. $t \in$ *set* (*parts_in_layer t1 t2 i*) $\Longrightarrow k + c *$ *rank t* $\leq k + c *$ *rank t2*
using *part_rank_le_snd assms*(2,5) *mult_left_mono* **by** *fastforce*
have $(\sum t \leftarrow$ *parts_in_layer t1 t2 i*. $k + c *$ *rank t*$)$
$\leq (\sum \_ \leftarrow$ *parts_in_layer t1 t2 i*. $k + c *$ *rank t2*$)$
using *each* **by** (*simp add*: *sum_list_mono*)
also have ... = *real* (*length* (*parts_in_layer t1 t2 i*)) $* (k + c *$ *rank t2*$)$
by (*simp add*: *sum_list_triv*)
also have ... $\leq$ *real C* $* (k + c *$ *rank t2*$)$
using *length_parts_in_layer_le_C rank_pos assms*
by (*auto intro*: *add_nonneg_nonneg mult_right_mono*)
finally show *?thesis* **.**
qed

Thus the coarsening loses only a constant factor. Below a rank root, *t1* has at most C further nodes in the layer (*layer_bound_const*), and every part collected there is a further split of the recorded part, hence of no larger rank. Per layer, the weighted part sum is therefore at most C times the root-part sum.

lemma *parts_rank_le_C_root_parts_rank*:
assumes *inv t1 inv t2* $k > 0$ $c > 0$
shows $(\sum t \leftarrow$ *parts_in_layer t1 t2 i*. $k + c *$ *rank t*$)$
$\leq$ *real C* $* (\sum t \leftarrow$ *rank_root_parts_in_layer t1 t2 i*. $k + c *$ *rank t*$)$
using *assms*(1,2)
proof (*induction t1 arbitrary*: *t2 rule*: *tree2_induct*)
case *Leaf*
then show *?case*
by (*simp add*: *parts_in_layer.simps rank_root_parts_in_layer.simps*)
next
case (*Node l1 a bx r1*)
let *?t1* = *Node l1* (*a*, *bx*) *r1*
obtain *l2 b r2* **where** *SPL*: *split a t2* = (*l2*, *b*, *r2*)
by (*cases split a t2*) *auto*
consider (*leaf*) *t2* = *Leaf*
| (*band*) *t2* $\neq$ *Leaf in_band i* (*rank ?t1*)
| (*rec*) *t2* $\neq$ *Leaf* $\neg$ *in_band i* (*rank ?t1*)
by *auto*
then show *?case*

```
  proof cases
    case leaf
    then show ?thesis
      by (simp add: parts_in_layer.simps rank_root_parts_in_layer.simps)
  next
    case band
    then show ?thesis
    using sum_parts_band_le_C[OF rule_bal[OF Node.prems(1)] Node.prems(2)
band(2) assms(3,4)]
      by (simp add: rank_root_parts_in_layer.simps[of ?t1 t2] SPL)
  next
    case rec
    have (∑ t ← parts_in_layer l1 l2 i. k + c * rank t)
          ≤ real C * (∑ t ← rank_root_parts_in_layer l1 l2 i. k + c * rank t)
      and (∑ t ← parts_in_layer r1 r2 i. k + c * rank t)
          ≤ real C * (∑ t ← rank_root_parts_in_layer r1 r2 i. k + c * rank t)
      using Node.IH Node.prems SPL by(auto simp: split_inv dest!: inv_Node)
    then show ?thesis
      using rec
    by (auto simp: parts_in_layer.simps[of ?t1 t2] rank_root_parts_in_layer.simps[of
?t1 t2]
                   SPL distrib_left)
  qed
qed
```

Combining the layering identity with the per-layer approximation yields the layered root-part sum.

```
corollary sum_parts_le_C_root:
  assumes inv t1 inv t2 k > 0 c > 0
  shows (∑ t←split_parts t1 t2. k + c * rank t)
      ≤ real C * (∑ i ≤ nat ⌈rank t1⌉.
          ∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t)
proof −
  have (∑ t←split_parts t1 t2. k + c * rank t)
     = (∑ i ≤ nat ⌈rank t1⌉. ∑ t←parts_in_layer t1 t2 i. k + c * rank t)
    using sum_parts_as_layers[OF rule_bal[OF assms(1)]] by simp
  also have ... ≤ (∑ i ≤ nat ⌈rank t1⌉.
                   real C * (∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t))
       using parts_rank_le_C_root_parts_rank[OF assms] by (auto intro:
sum_mono)
  also have ... = real C * (∑ i ≤ nat ⌈rank t1⌉.
                    ∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t)
    by (simp add: sum_distrib_left)
  finally show ?thesis .
qed
```

5.2 Split-work as a layered double sum

With the above in place *split_work* can be rewritten into the target double-sum.

corollary *split_work_le_C_root_parts*:
assumes *inv t1 inv t2*
shows *split_work t1 t2*
$\leq$ *real C* $*$ *K_split* $*$ ($\sum i \leq$ *nat* $\lceil$*rank t1*$\rceil$.
$\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *2* $+$ *rank t*)
proof $-$
have *split_work t1 t2*
$\leq$ ($\sum t \leftarrow$ *split_parts t1 t2*. *2* $+$ *K_split* $*$ *rank t*)
using *split_work_le_part_sum*[*OF assms*(*2*)] **.**
also have ... $\leq$ ($\sum t \leftarrow$ *split_parts t1 t2*. *K_split* $*$ *2* $+$ *K_split* $*$ *rank t*)
using *K_split_ge_1* **by** (*auto intro*: *sum_list_mono*)
also have ... $\leq$ *real C* $*$ ($\sum i \leq$ *nat* $\lceil$*rank t1*$\rceil$.
$\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *K_split* $*$ *2* $+$ *K_split* $*$ *rank t*)
using *sum_parts_le_C_root*[*OF assms*, *of K_split* $*$ *2 K_split*] *K_split_pos*
by *simp*
also have ... $=$ *real C* $*$ *K_split* $*$ ($\sum i \leq$ *nat* $\lceil$*rank t1*$\rceil$.
$\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *2* $+$ *rank t*)
proof $-$
have ($\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *K_split* $*$ *2* $+$ *K_split* $*$ *rank t*)
$=$ *K_split* $*$ ($\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *2* $+$ *rank t*) **for** *i*
unfolding *distrib_left*[*symmetric*] **using** *sum_list_const_mult* **by** *fast*
then show *?thesis*
by (*simp add*: *sum_distrib_left algebra_simps*)
qed
finally show *?thesis* **.**
qed

Only non-empty layers contribute.

definition *L* **where** *L t1 t2* $=$ {*i*. *i* $\leq$ *nat* $\lceil$*rank t1*$\rceil$ $\wedge$ *rank_root_parts_in_layer t1 t2 i* $\neq$ []}

corollary *L_finite*: *finite* (*L t1 t2*)
using *finite_subset* **by** (*auto simp*: *L_def*)

lemma *sum_layers_nonempty*:
($\sum i \leq$ *nat* $\lceil$*rank t1*$\rceil$. $\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *f t*) $=$
($\sum i \in$ *L t1 t2*. $\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *f t*)
by (*auto simp*: *L_def atMost_def intro*!: *sum.mono_neutral_right*)

corollary *split_work_le_C_root_parts_L*:
assumes *inv t1 inv t2*
shows *split_work t1 t2*
$\leq$ *real C* $*$ *K_split* $*$ ($\sum i \in$ *L t1 t2*.
$\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. *2* $+$ *rank t*)

```
proof −
  have split_work t1 t2
      ≤ real C * K_split * (∑ i ≤ nat ⌈rank t1⌉.
          ∑ t←rank_root_parts_in_layer t1 t2 i. 2 + rank t)
    using split_work_le_C_root_parts[OF assms] .
  also have ... = real C * K_split * (∑ i ∈ L t1 t2.
          ∑ t←rank_root_parts_in_layer t1 t2 i. 2 + rank t)
    using sum_layers_nonempty by auto
  finally show ?thesis .
qed
```

5.3 Inner sum analysis

In this subsection a closed form of the inner sum is obtained. s denotes the number of rank root parts in a specific layer which inherits the geometric decay of the rank roots.

```
definition s where s t1 t2 i = length (rank_root_parts_in_layer t1 t2 i)

lemma s_le_geometric:
  assumes inv t1
  shows real (s t1 t2 i) ≤ size1 t1 / 2 powr (i / c_u)
proof −
  have real (s t1 t2 i) ≤ real (length (rank_roots t1 i))
    using length_root_parts_le_rank_roots by (simp add: s_def)
  also have ... ≤ size1 t1 / 2 powr (i / c_u)
    using rank_roots_count_powr_le assms by blast
  finally show ?thesis .
qed
```

Independently of the layer, the number of root parts is also bounded by the size of the decomposed tree itself. Since every recorded part is non-empty, *sizes_root_layer* leaves room for at most *size t2* of them.

```
lemma s_le_size:
  assumes inv t2
  shows s t1 t2 i ≤ size t2
proof −
  let ?xs = rank_root_parts_in_layer t1 t2 i
  have ⋀t. t ∈ set ?xs ⟹ 1 ≤ size t
    using root_parts_nonempty by (metis eq_size_0 less_one not_le)
  then have (∑ t←?xs. 1) ≤ (∑ t←?xs. size t)
    by (intro sum_list_mono)
  then show ?thesis
    using sizes_root_layer[OF assms] dual_order.trans
    by (simp add: s_def sum_list_triv) blast
qed
```

Replace rank with $c_u * log\ 2\ (real\ (size1\ t))$ using *rank_le_cu_log_size1*.

```
lemma inner_sum_log_bound:
```

```
  assumes inv t2
  shows (∑ t←rank_root_parts_in_layer t1 t2 i. rank t)
      ≤ c_u * (∑ t←rank_root_parts_in_layer t1 t2 i. log 2 (size1 t))
proof −
  have ∀ t ∈ set (rank_root_parts_in_layer t1 t2 i). rank t ≤ c_u * log 2 (size1 t)
    using root_parts_inv[OF assms] rank_le_cu_log_size1' by blast
  then have sum_list (map rank (rank_root_parts_in_layer t1 t2 i))
        ≤ sum_list (map (λt. c_u * log 2 (size1 t)) (rank_root_parts_in_layer t1
t2 i))
    by (auto intro: sum_list_mono)
  then show ?thesis
    by (metis sum_list_const_mult)
qed

corollary inner_sum_log_bound_size:
  assumes inv t2
  shows (∑ t←rank_root_parts_in_layer t1 t2 i. rank t)
      ≤ c_u * (∑ t←rank_root_parts_in_layer t1 t2 i. log 2 (size t + 1))
  by (metis (lifting) ext assms inner_sum_log_bound size1_size)
```

Since the inner sum can now be written as a sum of logs, Jensen's inequality for concave functions on finite sets (c.f. *jensen_finite*) pulls the log out of the sum, replacing the sum of logs with a single log of the average. This is key because the size of an individual part in a layer cannot be known, but their combined size is bounded - the concavity of *log* makes the average the right quantity to work with.

```
lemma bound_sum_logs:
  assumes rank_root_parts_in_layer t1 t2 i ≠ []
  shows (∑ t←rank_root_parts_in_layer t1 t2 i. log 2 (real (size t + 1)))
      ≤ (s t1 t2 i) * log 2 ((∑ t ← rank_root_parts_in_layer t1 t2 i. size t + 1)
/ (s t1 t2 i))
proof −
  let ?xs = rank_root_parts_in_layer t1 t2 i
  let ?vals = map (λt. real (size t + 1)) ?xs
  let ?n = length ?xs
  let ?f = log 2
  let ?D = {0<..}

  have n_pos: ?n > 0 using assms by simp
  have pos: ⋀x. x ∈ set ?vals ⟹ x ∈ ?D by auto
  have conc: concave_on ?D ?f using log_concave by simp

  have jensen: (1 / real ?n) * sum_list (map ?f ?vals)
    ≤ ?f ((1 / real ?n) * sum_list ?vals)
    using jensen_list[OF _ conc pos, of ?vals] n_pos by simp
  have (∑ t←?xs. log 2 (real (size t + 1))) = sum_list (map ?f ?vals)
    by (metis (mono_tags, lifting) list.map_comp map_eq_conv o_apply)
 also have (1 / real ?n) * sum_list (map ?f ?vals) ≤ ?f ((1 / real ?n) * sum_list
?vals)
```

```
    using jensen by simp
  also have sum_list ?vals = real (∑ t←?xs. size t + 1)
        by (metis (mono_tags, lifting) list.map_comp map_eq_conv o_apply
sum_list_of_nat)
  finally have (1 / real ?n) * (∑ t←?xs. log 2 (real (size t + 1)))
    ≤ log 2 (real (∑ t←?xs. size t + 1) / real ?n)
    by simp
  then have (∑ t←?xs. log 2 (real (size t + 1)))
      ≤ real ?n * log 2 (real (∑ t←?xs. size t + 1) / real ?n)
    using n_pos by (simp add: divide_simps) argo
  moreover have real ?n = real (s t1 t2 i)
    by (simp add: s_def)
  moreover have (∑ t←?xs. size t + 1) = (∑ t←rank_root_parts_in_layer t1
t2 i. size t + 1)
    by blast
  ultimately show ?thesis
    by simp
qed
```

Combining the above with the size bound *sizes_root_layer* enables simplification of the log.

```
lemma layer_step_1:
  assumes inv t2 s t1 t2 i > 0
  shows (s t1 t2 i) * log 2 ((∑ t ← rank_root_parts_in_layer t1 t2 i. size t + 1)
/ (s t1 t2 i))
      ≤ (s t1 t2 i) * log 2 (size t2 / (s t1 t2 i) + 1)
proof −
  let ?xs = rank_root_parts_in_layer t1 t2 i
  let ?n = s t1 t2 i
  let ?sum = (∑ t ← ?xs. size t + 1)

  have npos: ?n > 0
    using assms by simp
  have split_sum: ?sum = (∑ t ← ?xs. size t) + ?n
    by (simp add: sum_list_Suc s_def)

  have ?sum / ?n ≤ size t2 / ?n + 1
  proof −
    have (∑ t ← ?xs. size t) ≤ size t2
      using sizes_root_layer[OF assms(1)] by blast
    then have ?sum ≤ size t2 + ?n
      using split_sum by linarith
    then have ?sum / ?n ≤ (size t2 + ?n) / ?n
      using npos by (auto intro: divide_right_mono)
    also have ... = size t2 / ?n + 1
      using npos by (simp add: add_divide_distrib)
    finally show ?thesis .
  qed
```

```
  then have log 2 (?sum / ?n) ≤ log 2 (size t2 / ?n + 1)
    using less_eq_real_def log_mono npos split_sum by auto
  then show ?thesis
    by (simp add: mult_left_mono npos)
qed
```

Combining the three steps above, the whole contribution of a layer is bounded by a term in the part count s alone.

```
lemma layer_term_le_s:
  assumes inv t2 i ∈ L t1 t2 0 ≤ c
  shows (∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t)
      ≤ (s t1 t2 i) * (k + c * c_u * log 2 (size t2 / (s t1 t2 i) + 1))
proof −
  let ?xs = rank_root_parts_in_layer t1 t2 i
  have spos: s t1 t2 i > 0 using assms(2) by (simp add: L_def s_def)

  have (∑ t←?xs. rank t) ≤ c_u * (∑ t←?xs. log 2 (size t + 1))
    using inner_sum_log_bound_size[OF assms(1)] .
  also have ... ≤ c_u * ((s t1 t2 i) * log 2 ((∑ t ← ?xs. size t + 1) / (s t1 t2 i)))
  proof −
    have ?xs ≠ [] using spos s_def by auto
      then show ?thesis using bound_sum_logs c_vals by (simp add:
mult_left_mono)
  qed
  also have ... ≤ c_u * ((s t1 t2 i) * log 2 (size t2 / (s t1 t2 i) + 1))
   using layer_step_1[OF assms(1) spos] c_vals by (simp add: mult_left_mono)
  finally have rank_chain: (∑ t←?xs. rank t)
      ≤ c_u * ((s t1 t2 i) * log 2 (size t2 / (s t1 t2 i) + 1)) .

  have (∑ t←?xs. k + c * rank t) = k * (s t1 t2 i) + c * (∑ t←?xs. rank t)
    by (simp add: sum_list_addf s_def sum_list_triv sum_list_const_mult alge-
bra_simps)
  also have ... ≤ k * (s t1 t2 i) + c * (c_u * ((s t1 t2 i) * log 2 (size t2 / (s t1
t2 i) + 1)))
    using mult_left_mono[OF rank_chain assms(3)] by linarith
  also have ... = (s t1 t2 i) * (k + c * c_u * log 2 (size t2 / (s t1 t2 i) + 1))
    by argo
  finally show ?thesis .
qed
```

From strict monotonicity of $x \log_2 \left(1 + \frac{n}{x}\right)$ (c.f. *xlog_term_mono*) s can be substituted with its geometric upper bound.

```
lemma layer_term_le_geometric:
  assumes inv t1 inv t2 size t2 > 0 i ∈ L t1 t2 and k0: 0 ≤ k and c0: 0 ≤ c
  shows (∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t)
      ≤ (size1 t1 / 2 powr (i / c_u)) *
          (k + c * c_u * log 2 (size t2 / (size1 t1 / 2 powr (i / c_u)) + 1))
proof −
  have spos: 0 < real (s t1 t2 i) using assms(4) by (simp add: L_def s_def)
```

```
  have npos: 0 < real (size t2) using assms(3) by simp
  have ccu: 0 ≤ c * c_u using c0 c_u_nonneg by (rule mult_nonneg_nonneg)

  have (∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t)
     ≤ (s t1 t2 i) * (k + c * c_u * log 2 (size t2 / (s t1 t2 i) + 1))
    using layer_term_le_s[OF assms(2,4) c0] .
  also have ... ≤ (size1 t1 / 2 powr (i / c_u)) *
          (k + c * c_u * log 2 (size t2 / (size1 t1 / 2 powr (i / c_u)) + 1))
    using xlog_term_mono[OF spos s_le_geometric[OF assms(1)] npos k0 ccu] .
  finally show ?thesis .
qed
```

5.4 Closed-form work bound

Having obtained a closed form for the inner sum, the rest of the proof proceeds purely analytically. All previous steps are combined to show that each layer's contribution is bounded by a term involving *size1* $t_1/2^{i/c_u}$ (the geometric decay) and a logarithmic factor. The sum index i only appears as an exponent in the denominator, which is key for obtaining the final closed form via geometric series.

```
corollary dsum_le_sum_logs:
  assumes inv t1 inv t2 size t2 > 0 and k0: 0 ≤ k and c0: 0 ≤ c
  shows (∑ i ∈ L t1 t2. ∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t)
      ≤ (∑ i ∈ L t1 t2.
           (size1 t1 / 2 powr (i / c_u)) *
           (k + c * c_u * log 2 (size t2 / (size1 t1 / 2 powr (i / c_u)) + 1)))
  using assms layer_term_le_geometric sum_mono by meson
```

The sum from *dsum_le_sum_logs* has summands of the form $z_i \cdot (k + c \cdot c_u \log_2(n/z_i + 1))$ with $z_i = m/2^{i/c_u}$. After rewriting z_i in terms of the geometric ratio $1/2^{1/c_u}$, the log is split and linearized pointwise, and the sum falls apart into geometric and arithmetic-geometric series, bounded by S_1 and S_2 respectively.

```
lemma final_sum_decomposed_ge_pivot:
  assumes sz2: size t2 > 0 and k0: 0 ≤ k and c0: 0 ≤ c
  shows (∑ i ∈ L t1 t2.
           (size1 t1 / 2 powr (i / c_u)) *
           (k + c * c_u * log 2 (size t2 / (size1 t1 / 2 powr (i / c_u)) + 1)))
      ≤ size1 t1 * (k * S1 c_u + c * c_u * S1 c_u + c * S2 c_u)
       + c * c_u * S1 c_u * size1 t1 * log 2 (size t2 / size1 t1 + 1)
proof −
  let ?A = L t1 t2
  let ?m = real (size1 t1)
  let ?n = real (size t2)
  let ?q = geom_ratio c_u

  have cu_pos: c_u > 0 using c_vals by force
  have finA: finite ?A using L_finite .
```

```
  have m_pos: ?m > 0 by simp
  have nm_pos: ?n / ?m > 0 using sz2 m_pos by simp
  have q_ge0: ?q ^ i ≥ 0 for i using geom_ratio_range[OF cu_pos] by simp

  have inv_powr_as_q: 1 / (2 powr (real i / c_u)) = ?q ^ i for i
    using inv_powr_geom_ratio[OF cu_pos] .

  have summand_rw:
    ((?m / (2 powr (real i / c_u))) *
      (k + c * c_u * log 2 (?n / (?m / (2 powr (real i / c_u))) + 1)))
     = ?m * (?q ^ i) * (k + c * c_u * log 2 ((?n / ?m) * (2 powr (real i / c_u)) +
1)) for i
  proof −
    have ?m / (2 powr (real i / c_u)) = ?m * (?q ^ i)
      by (metis inv_powr_as_q mult.right_neutral times_divide_eq_right)
     moreover have ?n / (?m / (2 powr (real i / c_u))) = (?n / ?m) * (2 powr
(real i / c_u))
      using m_pos by force
    ultimately show ?thesis by presburger
  qed

  — Pointwise: split the log and linearize the power part
  have pw: log 2 ((?n / ?m) * (2 powr (real i / c_u)) + 1)
        ≤ log 2 (?n / ?m + 1) + (1 + real i / c_u) for i
  proof −
    have log 2 ((?n / ?m) * (2 powr (real i / c_u)) + 1)
       ≤ log 2 (?n / ?m + 1) + log 2 (2 powr (real i / c_u) + 1)
      using log_split[of ?n / ?m 2 powr (real i / c_u)] nm_pos by auto
    then show ?thesis
      using log_2powr_plus_1_le[of real i / c_u] cu_pos by simp
  qed

  — Pointwise: distribute into the three series summands
  have expand: ?m * (?q ^ i) * (k + c * c_u * (log 2 (?n / ?m + 1) + (1 + real i
/ c_u)))
            = ?m * ((k + c * c_u) * ?q ^ i + c * c_u * ?q ^ i * log 2 (?n / ?m + 1)
                  + c * real i * ?q ^ i) for i
    using cu_pos by (simp add: field_simps)

  have (∑ i ∈ ?A.
        (?m / (2 powr (i / c_u))) *
        (k + c * c_u * log 2 (?n / (?m / (2 powr (i / c_u))) + 1)))
     = (∑ i ∈ ?A. ?m * (?q ^ i) * (k + c * c_u * log 2 ((?n / ?m) * (2 powr (real
i / c_u)) + 1)))
    using summand_rw by simp

  — Split the log and linearize inside the sum
  also have ... ≤ (∑ i ∈ ?A. ?m * (?q ^ i) *
     (k + c * c_u * (log 2 (?n / ?m + 1) + (1 + real i / c_u))))
```

using *pw m_pos q_ge0 c_vals c0*
by (*auto intro*!: *mult_nonneg_nonneg sum_mono mult_left_mono add_left_mono*)

— Distribute and split into three geometric series
also have ... = $(\sum i \in ?A.\ ?m * ((k + c * c_u) * ?q \hat{} i$
$+ c * c_u * ?q \hat{} i * log\ 2\ (?n\ /\ ?m + 1) + c * real\ i * ?q \hat{} i))$
by (*simp add*: *expand*)

also have ... = $?m * ($
$(k + c * c_u) * (\sum i \in ?A.\ ?q \hat{} i)$
$+ c * c_u * (\sum i \in ?A.\ ?q \hat{} i) * log\ 2\ (?n\ /\ ?m + 1)$
$+ c * (\sum i \in ?A.\ real\ i * (?q \hat{} i)))$
by (*simp add*: *sum.distrib sum_distrib_left sum_distrib_right algebra_simps*)

also have ... $\leq ?m * ($
$(k + c * c_u) * S1\ c_u$
$+ c * c_u * S1\ c_u * log\ 2\ (?n\ /\ ?m + 1)$
$+ c * S2\ c_u)$
using *geom_ratio_sum_le*[*OF cu_pos finA*] *geom_ratio_arith_sum_le*[*OF cu_pos finA*] *m_pos nm_pos c_vals k0 c0*
by (*auto simp*: *add_pos_pos*
intro!: *mult_left_mono add_mono mult_right_mono add_nonneg_nonneg mult_nonneg_nonneg*)

— Gather terms
also have ... = $?m * (k * S1\ c_u + c * c_u * S1\ c_u + c * S2\ c_u)$
$+ c * c_u * S1\ c_u * ?m * log\ 2\ (?n\ /\ ?m + 1)$
by *argo*
finally show *?thesis* **by** *simp*
qed

The bound of *final_sum_decomposed_ge_pivot* is tight only while the decomposed tree *t2* is at least as large as the pivot tree *t1*. Otherwise the geometric bound $m/2^{i/c_u}$ on the layer counts exceeds $n = size\ t_2$ in the upper layers and the linear term degenerates to $O(m)$. This is the second case in the proof of [4, Theorem 11] - the layers are cut at the threshold $\theta = c_u \log_2(m/n)$, where the two bounds on s coincide.

- Layers $i \leq \theta$ are bounded via *s_le_size*. Each contributes at most $n(k + c\,c_u)$ and there are at most $\theta + 1$ of them.
- Layers $i > \theta$ are bounded geometrically as before, but re-indexed relative to the first such layer $i_0 = \lfloor \theta \rfloor + 1$. There the geometric bound has already decayed to n, so the sum becomes a geometric series in n.

lemma *final_sum_decomposed_le_pivot*:
assumes *inv t1 inv t2 size t2 > 0 size t2 ≤ size1 t1*
and *k0*: $0 \leq k$ **and** *c0*: $0 \leq c$

shows $(\sum i \in L\ t1\ t2.\ \sum t{\leftarrow}rank\_root\_parts\_in\_layer\ t1\ t2\ i.\ k + c * rank\ t)$
$\leq size\ t2 * ((k + c * c_u) * (1 + S1\ c_u) + c * S1\ c_u + c * S2\ c_u)$
$+ (k + c * c_u) * c_u * size\ t2 * log\ 2\ (size1\ t1\ /\ size\ t2 + 1)$
proof −
let $?m = real\ (size1\ t1)$
let $?n = real\ (size\ t2)$
let $?\vartheta = c_u * log\ 2\ (?m\ /\ ?n)$
let $?i_0 = nat\ \lfloor ?\vartheta \rfloor + 1$
let $?L_1 = \{i \in L\ t1\ t2.\ i < ?i_0\}$
let $?L_2 = \{i \in L\ t1\ t2.\ ?i_0 \leq i\}$
let $?q = geom\_ratio\ c_u$
let $?term = \lambda i.\ (\sum t{\leftarrow}rank\_root\_parts\_in\_layer\ t1\ t2\ i.\ k + c * rank\ t)$

have cu_pos: $c_u > 0$ **using** c_vals **by** *force*
have cu_ne: $c_u \neq 0$ **using** cu_pos **by** *linarith*
have ccu: $0 \leq c * c_u$ **using** $c0\ c\_u\_nonneg$ **by** *fastforce*
have m_pos: $?m > 0$ **by** *simp*
have n_pos: $?n > 0$ **using** $assms(3)$ **by** *simp*
have mn_pos: $?m\ /\ ?n > 0$ **using** $m\_pos\ n\_pos$ **by** *simp*
have mn_ge1: $1 \leq ?m\ /\ ?n$ **using** $assms(4)\ n\_pos$ **by** *force*
have ϑ_nonneg: $0 \leq ?\vartheta$ **using** $cu\_pos\ mn\_ge1\ mn\_pos$ **by** *simp*
have $i0\_bounds$: $?\vartheta < real\ ?i_0\ real\ ?i_0 \leq ?\vartheta + 1$
using ϑ_nonneg **by** *linarith*+

— Split the layers at the threshold
have $(\sum i \in L\ t1\ t2.\ ?term\ i) = (\sum i \in ?L_1 \cup ?L_2.\ ?term\ i)$
by $(rule\ sum.cong)\ auto$
also have $\ldots = (\sum i \in ?L_1.\ ?term\ i) + (\sum i \in ?L_2.\ ?term\ i)$
by $(rule\ sum.union\_disjoint)\ (auto\ intro{:}\ rev\_finite\_subset[OF\ L\_finite])$
finally have sum_split: $(\sum i \in L\ t1\ t2.\ ?term\ i)$
$= (\sum i \in ?L_1.\ ?term\ i) + (\sum i \in ?L_2.\ ?term\ i)$ **.**

— Lower layers: at most $size\ t2$ parts each, and at most $\vartheta + 1$ such layers
have $L1\_term$: $?term\ i \leq ?n * (k + c * c_u)$ **if** $i \in L\ t1\ t2$ **for** i
proof −
have $spos$: $0 < real\ (s\ t1\ t2\ i)$ **using** $that$ **by** $(simp\ add{:}\ L\_def\ s\_def)$
have sle: $real\ (s\ t1\ t2\ i) \leq ?n$ **using** $s\_le\_size[OF\ assms(2)]$ **by** *simp*
have $?term\ i \leq (s\ t1\ t2\ i) * (k + c * c_u * log\ 2\ (?n\ /\ (s\ t1\ t2\ i) + 1))$
using $layer\_term\_le\_s[OF\ assms(2)\ that\ c0]$ **.**
also have $\ldots \leq ?n * (k + c * c_u * log\ 2\ (?n\ /\ ?n + 1))$
using $xlog\_term\_mono[OF\ spos\ sle\ n\_pos\ k0\ ccu]$ **.**
also have $\ldots = ?n * (k + c * c_u)$
using n_pos **by** *simp*
finally show $?thesis$ **.**
qed

have $card\_L1$: $real\ (card\ ?L_1) \leq ?\vartheta + 1$
proof −
have $card\ ?L_1 \leq card\ \{..<?i_0\}$

```
    by (intro card_mono) auto
  then have real (card ?L1) ≤ real ?i0
    by simp
  with i0_bounds(2) show ?thesis
    by linarith
qed

have sum_L1: (∑ i ∈ ?L1. ?term i) ≤ (?ϑ + 1) * (?n * (k + c * cu))
proof −
  have (∑ i ∈ ?L1. ?term i) ≤ (∑ i ∈ ?L1. ?n * (k + c * cu))
    using L1_term by (intro sum_mono) auto
  also have ... = real (card ?L1) * (?n * (k + c * cu))
    by simp
  also have ... ≤ (?ϑ + 1) * (?n * (k + c * cu))
    using card_L1 n_pos k0 ccu
          by (intro mult_right_mono) (auto intro!: mult_nonneg_nonneg
add_nonneg_nonneg)
  finally show ?thesis .
qed

— Upper layers: the geometric bound decays from size t2 onwards
have L2_term: ?term i ≤ ?n * (k + c * cu + c) * ?q ^ (i − ?i0)
                      + c * ?n * (real (i − ?i0) * ?q ^ (i − ?i0))
  if i ∈ ?L2 for i
proof −
  let ?j = i − ?i0
  let ?z = ?m / 2 powr (real i / cu)
  have iL: i ∈ L t1 t2 and i_ge: ?i0 ≤ i using that by auto

  have i_split: real i / cu = real ?i0 / cu + real ?j / cu
    using i_ge cu_pos by (simp add: field_simps)
  have pow_split: 2 powr (real i / cu) = 2 powr (real ?i0 / cu) * 2 powr (real ?j
/ cu)
    by (simp add: i_split powr_add)

  — i0 lies within one layer above the threshold, i.e. m/n ≤ 2 powr (i0/cu) ≤
(m/n) 2 powr (1/cu)
  have pow_i0_lb: ?m / ?n ≤ 2 powr (real ?i0 / cu)
  proof −
    have log 2 (?m / ?n) < real ?i0 / cu
      using i0_bounds(1) cu_pos by (simp add: field_simps)
    then have 2 powr (log 2 (?m / ?n)) < 2 powr (real ?i0 / cu)
      by auto
    then show ?thesis
      using mn_pos by simp
  qed
  have pow_i0_ub: 2 powr (real ?i0 / cu) ≤ (?m / ?n) * 2 powr (1 / cu)
  proof −
    have real ?i0 / cu ≤ (?ϑ + 1) / cu
```

```
    using i0_bounds(2) cu_pos by (intro divide_right_mono) auto
  also have ... = log 2 (?m / ?n) + 1 / c_u
    using cu_ne by (simp add: add_divide_distrib)
  finally have 2 powr (real ?i_0 / c_u) ≤ 2 powr (log 2 (?m / ?n) + 1 / c_u)
    by simp
  then show ?thesis
    using mn_pos by (simp add: powr_add)
qed

— The geometric bound, re-indexed: z ≤ n q^j
have z_le: ?z ≤ ?n * ?q ^ ?j
proof −
  have (?m / ?n) * 2 powr (real ?j / c_u) ≤ 2 powr (real i / c_u)
    using pow_i0_lb unfolding pow_split by (intro mult_right_mono) auto
  then have ?z ≤ ?m / ((?m / ?n) * 2 powr (real ?j / c_u))
    using mn_pos by (intro divide_left_mono mult_pos_pos) auto
  also have ... = ?n / 2 powr (real ?j / c_u)
    using m_pos n_pos by (simp add: field_simps)
  also have ... = ?n * ?q ^ ?j
        by (metis inv_powr_geom_ratio[OF cu_pos] mult.right_neutral
times_divide_eq_right)
  finally show ?thesis .
qed

— The logarithmic factor: n / z ≤ 2 powr ((j + 1) / c_u)
have log_le: log 2 (?n / ?z + 1) ≤ 1 + (real ?j + 1) / c_u
proof −
  have ?n / ?z = (?n / ?m) * 2 powr (real ?i_0 / c_u) * 2 powr (real ?j / c_u)
    using m_pos unfolding pow_split by fastforce
  also have ... ≤ (?n / ?m) * ((?m / ?n) * 2 powr (1 / c_u)) * 2 powr (real
?j / c_u)
        using pow_i0_ub n_pos m_pos by (intro mult_right_mono
mult_left_mono) auto
  also have ... = 2 powr (1 / c_u) * 2 powr (real ?j / c_u)
    using n_pos m_pos by auto
  also have ... = 2 powr ((real ?j + 1) / c_u)
    by (simp add: add_divide_distrib powr_add)
  finally have ?n / ?z ≤ 2 powr ((real ?j + 1) / c_u) .
  then have log 2 (?n / ?z + 1) ≤ log 2 (2 powr ((real ?j + 1) / c_u) + 1)
    by (auto intro!: log_mono add_nonneg_pos)
  also have ... ≤ 1 + (real ?j + 1) / c_u
    using cu_pos by (simp add: log_2powr_plus_1_le)
  finally show ?thesis .
qed

have lognn: 0 ≤ log 2 (?n / ?z + 1)
  by (simp add: log_def)

have ?term i ≤ ?z * (k + c * c_u * log 2 (?n / ?z + 1))
```

```
    using layer_term_le_geometric[OF assms(1,2,3) iL k0 c0] .
  also have ... ≤ (?n * ?q ^ ?j) * (k + c * c_u * (1 + (real ?j + 1) / c_u))
   using mult_left_mono[OF log_le ccu] k0 mult_nonneg_nonneg[OF ccu lognn]
     k0 mult_nonneg_nonneg[OF ccu lognn] z_le n_pos geom_ratio_range[OF
cu_pos]
    by(intro mult_mono) auto
  also have ... = ?n * (k + c * c_u + c) * ?q ^ ?j + c * ?n * (real ?j * ?q ^ ?j)
  proof −
    have eq: c * c_u * (1 + (real ?j + 1) / c_u) = c * c_u + c * (real ?j + 1)
      using cu_ne by (simp add: field_simps)
    show ?thesis
      by (subst eq) (simp add: algebra_simps)
  qed
  finally show ?thesis .
 qed

 have sum_L2: (∑ i ∈ ?L_2. ?term i) ≤ ?n * (k + c * c_u + c) * S1 c_u + c * ?n
* S2 c_u
 proof −
  let ?J = (λi. i − ?i_0) ‘ ?L_2
  have inj: inj_on (λi. i − ?i_0) ?L_2
    by (auto simp: inj_on_def eq_diff_iff)
  have finJ: finite ?J
    by (auto intro: rev_finite_subset[OF L_finite])
  have (∑ i ∈ ?L_2. ?term i)
     ≤ (∑ i ∈ ?L_2. ?n * (k + c * c_u + c) * ?q ^ (i − ?i_0)
                  + c * ?n * (real (i − ?i_0) * ?q ^ (i − ?i_0)))
    using L2_term by (auto intro!: sum_mono)
  also have ... = (∑ j ∈ ?J. ?n * (k + c * c_u + c) * ?q ^ j + c * ?n * (real j
* ?q ^ j))
    by (subst sum.reindex[OF inj]) (simp add: o_def)
  also have ... = ?n * (k + c * c_u + c) * (∑ j ∈ ?J. ?q ^ j)
             + c * ?n * (∑ j ∈ ?J. real j * ?q ^ j)
    by (simp add: sum.distrib sum_distrib_left)
  also have ... ≤ ?n * (k + c * c_u + c) * S1 c_u + c * ?n * S2 c_u
     using geom_ratio_sum_le[OF cu_pos finJ] geom_ratio_arith_sum_le[OF
cu_pos finJ] n_pos k0 c0 ccu
    by (auto intro!: add_mono mult_left_mono)
  finally show ?thesis .
 qed

 — Assemble; ϑ is absorbed into the logarithmic term
 have (∑ i ∈ L t1 t2. ?term i)
    ≤ (?ϑ + 1) * (?n * (k + c * c_u)) + (?n * (k + c * c_u + c) * S1 c_u + c *
?n * S2 c_u)
  using sum_split sum_L1 sum_L2 by linarith
 also have ... = ?n * ((k + c * c_u) * (1 + S1 c_u) + c * S1 c_u + c * S2 c_u)
           + (k + c * c_u) * c_u * ?n * log 2 (?m / ?n)
  by argo
```

also have ... $\le$ *?n* $*$ ((k + c $*$ c_u) $*$ (*1* + *S1* c_u) + c $*$ *S1* c_u + c $*$ *S2* c_u)
+ (k + c $*$ c_u) $*$ c_u $*$ *?n* $*$ *log 2* (*?m* / *?n* + *1*)
using *mn_pos k0 ccu cu_pos* **by** (*auto intro*!: *log_mono add_left_mono mult_left_mono*)
finally show *?thesis* **.**
qed

5.5 Unified closed form

The two closed forms combine into a single bound of the shape $A \cdot m + B \cdot m \log_2(n/m+1)$ in the smaller size m and the larger size n. For $m =$ *size1* t_1 the first form applies, and its logarithm only grows when *size t2* is replaced by *size1 t2*. For $m =$ *size1* t_2 the second form applies, and *xlog_mono* lets *size t2* grow to *size1 t2* in both factors at once. The constants are the pointwise maxima of the two cases.

lemma *S1_nonneg*: *0* $\le$ *S1* c_u **and** *S2_nonneg*: *0* $\le$ *S2* c_u
using *geom_ratio_sum_le*[*of* c_u] *geom_ratio_arith_sum_le*[*of* c_u] *c_vals* **by** *force*+

abbreviation *bound_mn* :: *real* $\Rightarrow$ *real* $\Rightarrow$ (*'a*$*$*'b*) *tree* $\Rightarrow$ (*'a*$*$*'b*) *tree* $\Rightarrow$ *real* **where**
bound_mn k c ta tb $\equiv$
size_min ta tb $*$ ((k + c $*$ c_u) $*$ (*1* + *S1* c_u) + c $*$ *S1* c_u + c $*$ *S2* c_u)
+ c_u $*$ (k + c $*$ c_u + c $*$ *S1* c_u) $*$ *size_min ta tb*
$*$ *log 2* (*size_max ta tb* / *size_min ta tb* + *1*)

lemma *final_sum_bound_mn*:
assumes *inv t1 inv t2 size t2* > *0* **and** *k0*: *0* $\le$ k **and** *c0*: *0* $\le$ c
shows ($\sum i \in L$ *t1 t2*. $\sum t \leftarrow$*rank_root_parts_in_layer t1 t2 i*. k + c $*$ *rank t*)
$\le$ *bound_mn k c t1 t2*
proof −
let *?A* = (k + c $*$ c_u) $*$ (*1* + *S1* c_u) + c $*$ *S1* c_u + c $*$ *S2* c_u
let *?B* = c_u $*$ (k + c $*$ c_u + c $*$ *S1* c_u)
have *ccu*: *0* $\le$ c $*$ c_u **using** *c0 c_u_nonneg* **by** *fastforce*
have *cS1*: *0* $\le$ c $*$ *S1* c_u **using** *c0 S1_nonneg* **by** *force*
have *A_nonneg*: *0* $\le$ *?A*
using *k0 c0 ccu S1_nonneg S2_nonneg* **by** *auto*
have *B_nonneg*: *0* $\le$ *?B*
using *k0 ccu cS1 c_u_nonneg* **by** *auto*

— The unified constants dominate those of both cases
have *A_ge*: k $*$ *S1* c_u + c $*$ c_u $*$ *S1* c_u + c $*$ *S2* c_u $\le$ *?A*
using *k0 ccu cS1* **by** *argo*
have *B_ge1*: c $*$ c_u $*$ *S1* c_u $\le$ *?B*
by (*metis add_increasing c_u_nonneg ccu k0 le_add_same_cancel2 mult.commute mult.left_commute mult_left_mono*)
have *B_ge2*: (k + c $*$ c_u) $*$ c_u $\le$ *?B*
by (*metis mult_left_mono le_add_same_cancel2 add.commute c_u_nonneg cS1 mult.commute*)

show *?thesis*
proof (*cases size t1 ≤ size t2*)
case *True*
then have *le*: *size1 t1 ≤ size1 t2* **by** (*simp add*: *size1_size*)
have *mn*: *size_min t1 t2 = size1 t1 size_max t1 t2 = size1 t2*
using *le* **by** (*simp_all add*: *min_absorb1 max_absorb2*)

have *lin*: *size1 t1 ∗ (k ∗ S1 c_u + c ∗ c_u ∗ S1 c_u + c ∗ S2 c_u) ≤ size1 t1 ∗ ?A*
using *A_ge* **by** (*auto intro*: *mult_left_mono*)
have *size t2 / size1 t1 ≤ size1 t2 / size1 t1*
by (*simp add*: *divide_right_mono size1_size*)
then have *lg*: *log 2 (size t2 / size1 t1 + 1) ≤ log 2 (size1 t2 / size1 t1 + 1)*
by (*intro log_mono*) (*auto intro*: *add_nonneg_pos*)
have *c ∗ c_u ∗ S1 c_u ∗ size1 t1 ∗ log 2 (size t2 / size1 t1 + 1)*
≤ ?B ∗ size1 t1 ∗ log 2 (size t2 / size1 t1 + 1)
using *B_ge1* **by** (*intro mult_right_mono*) (*auto simp*: *log_def*)
also have *... ≤ ?B ∗ size1 t1 ∗ log 2 (size1 t2 / size1 t1 + 1)*
using *lg B_nonneg* **by** (*auto intro*!: *mult_left_mono mult_nonneg_nonneg*)
finally have *lg′*: *c ∗ c_u ∗ S1 c_u ∗ size1 t1 ∗ log 2 (size t2 / size1 t1 + 1)*
≤ ?B ∗ size1 t1 ∗ log 2 (size1 t2 / size1 t1 + 1) **.**

have *($\sum$ i ∈ L t1 t2. $\sum$ t←rank_root_parts_in_layer t1 t2 i. k + c ∗ rank t)*
≤ ($\sum$ i ∈ L t1 t2. (size1 t1 / 2 powr (i / c_u)) ∗
(k + c ∗ c_u ∗ log 2 (size t2 / (size1 t1 / 2 powr (i / c_u)) + 1)))
using *dsum_le_sum_logs*[*OF assms*(*1*,*2*,*3*) *k0 c0*] **.**
also have *... ≤ size1 t1 ∗ (k ∗ S1 c_u + c ∗ c_u ∗ S1 c_u + c ∗ S2 c_u)*
+ c ∗ c_u ∗ S1 c_u ∗ size1 t1 ∗ log 2 (size t2 / size1 t1 + 1)
using *final_sum_decomposed_ge_pivot*[*OF assms*(*3*) *k0 c0*] **.**
also have *... ≤ size1 t1 ∗ ?A + ?B ∗ size1 t1 ∗ log 2 (size1 t2 / size1 t1 + 1)*
using *lin lg′* **by** *linarith*
finally show *?thesis* **using** *mn* **by** *simp*
next
case *False*
then have *le*: *size1 t2 ≤ size1 t1* **and** *sz*: *size t2 ≤ size1 t1*
by (*auto simp*: *size1_size*)
have *mn*: *size_min t1 t2 = size1 t2 size_max t1 t2 = size1 t1*
using *le* **by** (*auto simp*: *min_absorb2 max_absorb1*)

have *lin*: *size t2 ∗ ?A ≤ size1 t2 ∗ ?A*
using *A_nonneg* **by** (*simp add*: *mult_right_mono size1_size*)
have *size t2 ∗ log 2 (size1 t1 / size t2 + 1) ≤ size1 t2 ∗ log 2 (size1 t1 / size1 t2 + 1)*
using *assms*(*3*) **by** (*intro xlog_mono*) (*auto simp*: *size1_size*)
then have *(k + c ∗ c_u) ∗ c_u ∗ (size t2 ∗ log 2 (size1 t1 / size t2 + 1))*
≤ (k + c ∗ c_u) ∗ c_u ∗ (size1 t2 ∗ log 2 (size1 t1 / size1 t2 + 1))
using *k0 ccu c_u_nonneg* **by** (*auto intro*!: *mult_left_mono*)
also have *... ≤ ?B ∗ (size1 t2 ∗ log 2 (size1 t1 / size1 t2 + 1))*

```
    using B_ge2 log_def by (intro mult_right_mono) auto
  finally have lg: (k + c * c_u) * c_u * (size t2 * log 2 (size1 t1 / size t2 + 1))
      ≤ ?B * (size1 t2 * log 2 (size1 t1 / size1 t2 + 1)) .

  have (∑ i ∈ L t1 t2. ∑ t←rank_root_parts_in_layer t1 t2 i. k + c * rank t)
      ≤ size t2 * ?A + (k + c * c_u) * c_u * size t2 * log 2 (size1 t1 / size t2 + 1)
    using final_sum_decomposed_le_pivot[OF assms(1,2,3) sz k0 c0] .
  also have ... = size t2 * ?A + (k + c * c_u) * c_u * (size t2 * log 2 (size1 t1
/ size t2 + 1))
    by auto
  also have ... ≤ size1 t2 * ?A + ?B * (size1 t2 * log 2 (size1 t1 / size1 t2 +
1))
    using lin lg by linarith
  also have ... = size1 t2 * ?A + ?B * size1 t2 * log 2 (size1 t1 / size1 t2 +
1)
    by auto
  finally show ?thesis using mn by simp
 qed
qed
```

Putting all of the above together yields the final theorem [4, Theorem 11].

```
theorem split_work_bound_exact:
 assumes inv t1 inv t2 size t2 > 0
 shows split_work t1 t2 ≤ real C * K_split * bound_mn 2 1 t1 t2
proof −
 have split_work t1 t2
     ≤ real C * K_split * (∑ i ∈ L t1 t2.
         ∑ t←rank_root_parts_in_layer t1 t2 i. 2 + rank t)
   using split_work_le_C_root_parts_L[OF assms(1,2)] .
 also have ... ≤ real C * K_split * bound_mn 2 1 t1 t2
   using final_sum_bound_mn[where k=2 and c=1, OF assms] K_split_pos
C_pos
   by (auto intro: mult_left_mono)
 finally show ?thesis .
qed
```

6 Generic work-bound theorem

The three operations union, intersection and difference share one analytical chain. Writing (ta, tb) for the role pair - $(t1, t2)$ for union and intersection, $(t2, t1)$ for difference - each operation only has to provide two facts:

- the join-work bound $jw \le (\sum t{\leftarrow}split\_parts\ ta\ tb.\ k + c * rank\ t)$
- a decomposition $W \le jw + split\_work\ ta\ tb$

The former implies that the join-work is asymptotically no more than the split-work [4, Theorem 10]. The operation-specific theorems are mere instan-

tiations of this proof chain. First, observe that the argument for *split_work* is really an argument about summing *split_parts*.

corollary *part_sum_le_mn*:
assumes *inv ta inv tb size tb > 0*
assumes *kc*: *0 < k 0 < c*
shows $(\sum t \leftarrow$ *split_parts ta tb*. *k + c * rank t*$) \leq$ *real C * bound_mn k c ta tb*
proof −
have $(\sum t \leftarrow$ *split_parts ta tb*. *k + c * rank t*$)$
$\leq$ *real C * (*$\sum i \in$ *L ta tb*. $\sum t \leftarrow$*rank_root_parts_in_layer ta tb i*. *k + c * rank t*$)$
using *sum_parts_le_C_root sum_layers_nonempty assms* **by** *metis*
also have *... ≤ real C * bound_mn k c ta tb*
using *final_sum_bound_mn*[*OF assms*(*1*,*2*,*3*)] *kc C_pos* **by** (*auto intro*: *mult_left_mono*)
finally show *?thesis* **.**
qed

The arguments k, c of the packaged constants mirror the shape of the join-work bound. The shifts by *1 + 2 * K_split* resp. *K_split* account for the split-work and for the one unit of work charged per call.

definition *K_lin* :: *real ⇒ real ⇒ real* **where**
*K_lin k c = real C * ((k + 1 + 2 * K_split + (c + K_split) * c*$_u$*) * (1 + S1 c*$_u$*)*
*+ (c + K_split) * S1 c*$_u$ *+ (c + K_split) * S2 c*$_u$*)*
definition *K_log* :: *real ⇒ real ⇒ real* **where**
*K_log k c = real C * c*$_u$ ** (k + 1 + 2 * K_split + (c + K_split) * c*$_u$ *+ (c + K_split) * S1 c*$_u$*)*

lemma *K_lin_nonneg*: *0 ≤ k ⟹ 0 ≤ c ⟹ 0 ≤ K_lin k c*
unfolding *K_lin_def* **using** *S1_nonneg S2_nonneg c_u_nonneg C_pos K_split_pos*
by (*fastforce intro*!: *add_nonneg_nonneg mult_nonneg_nonneg*)

lemma *K_log_nonneg*: *0 ≤ k ⟹ 0 ≤ c ⟹ 0 ≤ K_log k c*
unfolding *K_log_def* **using** *S1_nonneg c_u_nonneg C_pos K_split_pos*
by (*fastforce intro*!: *add_nonneg_nonneg mult_nonneg_nonneg*)

theorem *W_bound_generic*:
fixes *W jw k c* :: *real*
assumes *invs*: *inv ta inv tb size tb > 0*
assumes *decomp*: *W ≤ jw + split_work ta tb*
assumes *jw_piv*: *jw* $\leq (\sum t \leftarrow$ *split_parts ta tb*. *k + c * rank t*$)$
assumes *k_pos*: *0 < k* **and** *c_pos*: *0 < c*
shows *W ≤ real C * bound_mn (k + 2 * K_split) (c + K_split) ta tb*
proof −
have *jw_bound*: *jw ≤ real C * bound_mn k c ta tb*
using *jw_piv part_sum_le_mn*[*OF invs k_pos c_pos*] **by** *linarith*
have *sw*: *split_work ta tb ≤ real C * K_split * bound_mn 2 1 ta tb*
using *split_work_bound_exact*[*OF invs*] **.**

have $W \leq$ *real* $C *$ *bound_mn* k c *ta tb* + *real* $C *$ $K\_split *$ *bound_mn 2 1 ta tb*
using *decomp jw_bound sw* **by** *linarith*
also have ... = *real* $C *$ *bound_mn* $(k + 2 * K\_split)$ $(c + K\_split)$ *ta tb*
by *algebra*
finally show *?thesis* **.**
qed

The concrete work functions charge one unit per call, base cases included. Since every inner call records exactly one part, this overhead amounts to *1* + *length* (*split_parts ta tb*) and is folded into the part sum by shifting k by one.

lemma *sum_list_parts_shift*:
$(\sum t \leftarrow xs.\ (k + 1) + c * rank\ t) = length\ xs + (\sum t \leftarrow xs.\ k + c * rank\ t)$
by (*induction xs*) (*simp_all add*: *algebra_simps*)

corollary *W_bound_generic′*:
fixes *W jw k c* :: *real*
assumes *invs*: *inv ta inv tb*
assumes *decomp*: $W \leq 1 +$ *length* (*split_parts ta tb*) + *jw* + *split_work ta tb*
assumes *jw_piv*: *jw* $\leq (\sum t \leftarrow$ *split_parts ta tb*. $k + c *$ *rank* $t)$
assumes *k_pos*: $0 < k$ **and** *c_pos*: $0 < c$
shows $W \leq 1 + K\_lin\ k\ c *$ *size_min ta tb*
$+ K\_log\ k\ c *$ (*size_min ta tb* $*$ *log* 2 (*size_max ta tb* / *size_min ta tb* + *1*))
proof (*cases tb* = *Leaf*)
case *True*
— An empty decomposed tree produces no parts, so only the unit of work of the call itself remains and the bound holds trivially
have *kc*: $0 \leq k\ 0 \leq c$
using *k_pos c_pos* **by** *linarith*+
have *parts*: *split_parts ta tb* = [] **and** *work*: *split_work ta tb* = *0*
by (*simp_all add*: *True split_parts.simps*[*of ta Leaf*] *split_work.simps*[*of ta Leaf*])
have $W \leq 1 +$ *jw* **and** *jw* ≤ 0
using *decomp jw_piv* **by** (*simp_all add*: *parts work*)
then have $W \leq 1$
by *linarith*
moreover have $0 \leq K\_lin\ k\ c *$ *size_min ta tb*
by (*intro mult_nonneg_nonneg K_lin_nonneg kc*) *simp*
moreover have $0 \leq K\_log\ k\ c *$ (*size_min ta tb* $*$ *log* 2 (*size_max ta tb* / *size_min ta tb* + *1*))
by (*rule mult_nonneg_nonneg*[*OF K_log_nonneg*[*OF kc*]]) (*simp add*: *log_def*)
ultimately show *?thesis*
by *linarith*
next
case *False*
then have *sz*: *size tb* > *0*
by (*cases tb*) *auto*

```
  have length (split_parts ta tb) + jw ≤ (∑ t ← split_parts ta tb. (k + 1) + c *
rank t)
    using jw_piv by (simp add: sum_list_parts_shift)
  moreover have 0 < k + 1
    using k_pos by linarith
  ultimately have length (split_parts ta tb) + jw + split_work ta tb
      ≤ real C * bound_mn (k + 1 + 2 * K_split) (c + K_split) ta tb
    using W_bound_generic[OF invs sz order.refl _ _ c_pos] by blast
  also have ... = K_lin k c * size_min ta tb
          + K_log k c * (size_min ta tb * log 2 (size_max ta tb / size_min ta tb
+ 1))
    unfolding K_lin_def K_log_def by algebra
  finally show ?thesis
    using decomp by linarith
qed
```

6.1 Asymptotic form

The closed-form bound above has the shape $1+O(m)+O(m\log_2(n/m+1))$. Since $1 \le m \le n$, the logarithm is at least $\log_2 2 = 1$, so both the constant and the linear part are absorbed and each operation runs in $O(m\log_2(n/m+1))$.

```
lemma K_lin_log_pos:
  assumes 0 ≤ k 0 ≤ c
  shows 0 < 1 + K_lin k c + K_log k c
  using K_lin_nonneg[OF assms] K_log_nonneg[OF assms] by linarith

lemma mlog_ge:
  fixes ta tb :: ('a * 'b) tree
  shows 1 ≤ size_min ta tb * log 2 (size_max ta tb / size_min ta tb + 1)
    and size_min ta tb ≤ size_min ta tb * log 2 (size_max ta tb / size_min ta tb
+ 1)
proof −
  have m1: 1 ≤ size_min ta tb and mM: size_min ta tb ≤ size_max ta tb
    by (simp_all add: size1_size min_le_iff_disj)
  then have 1 ≤ size_max ta tb / size_min ta tb
    by (simp add: pos_le_divide_eq)
  then have lg: 1 ≤ log 2 (size_max ta tb / size_min ta tb + 1)
    by (subst one_le_log_cancel_iff) linarith+
  have size_min ta tb * 1 ≤ size_min ta tb * log 2 (size_max ta tb / size_min
ta tb + 1)
    by (intro mult_left_mono) (use m1 lg in auto)
  then show size_min ta tb ≤ size_min ta tb * log 2 (size_max ta tb / size_min
ta tb + 1)
    by simp
  then show 1 ≤ size_min ta tb * log 2 (size_max ta tb / size_min ta tb + 1)
    using m1 by linarith
qed
```

corollary *W_bound_absorb*:
fixes *W jw k c* :: *real*
assumes *invs*: *inv ta inv tb*
assumes *decomp*: $W \leq 1 + length\ (split\_parts\ ta\ tb) + jw + split\_work\ ta\ tb$
assumes *jw_piv*: $jw \leq (\sum t \leftarrow split\_parts\ ta\ tb.\ k + c * rank\ t)$
assumes *k_pos*: $0 < k$ **and** *c_pos*: $0 < c$
shows $W \leq (1 + K\_lin\ k\ c + K\_log\ k\ c)$
$* (size\_min\ ta\ tb * log\ 2\ (size\_max\ ta\ tb\ /\ size\_min\ ta\ tb + 1))$
proof −
let *?g* = $size\_min\ ta\ tb * log\ 2\ (size\_max\ ta\ tb\ /\ size\_min\ ta\ tb + 1)$
have *lin*: $K\_lin\ k\ c * size\_min\ ta\ tb \leq K\_lin\ k\ c * ?g$
using *mlog_ge*(2) *K_lin_nonneg k_pos c_pos* **by** (*intro mult_left_mono*) *auto*
have $W \leq 1 + K\_lin\ k\ c * size\_min\ ta\ tb + K\_log\ k\ c * ?g$
using *W_bound_generic'*[*OF assms*] **.**
also have ... $\leq ?g + K\_lin\ k\ c * ?g + K\_log\ k\ c * ?g$
using *mlog_ge*(1)[*of ta tb*] *lin* **by** *linarith*
also have ... $= (1 + K\_lin\ k\ c + K\_log\ k\ c) * ?g$
by *algebra*
finally show *?thesis* **.**
qed

The formal argument uses the machinery defined in *HOL−Library.Going_To_Filter*. Intuitively, the filter defined below is generated by a collection of regions of trees i.e. for every threshold N the region A_N contains all invariant trees of size at least N. The regions are obviously nested i.e. $A_0 \supseteq A_1 \supseteq A_2 \supseteq \ldots$.

definition *tree_filter* :: ('*a* × '*b*) *tree filter* **where**
tree_filter = *size going_to at_top within* {*t*. *inv t*}

The connection to Big-O notation is the notion of *eventually*. A predicate P holds eventually in the filter iff P holds on some region A_N, i.e. iff there is a threshold N such that P holds for *every* invariant tree of size at least N. In other words, for all sufficiently large valid inputs.

lemma *eventually_tree_filter*: *eventually* (λ*t*. *inv t* ∧ 1 ≤ *size t*) *tree_filter*
proof −
have ∀ *t*. *inv t* ⟶ 1 ≤ *size t* ⟶ *inv t* ∧ 1 ≤ *size t*
by *fast*
then have *eventually* (λ*t*. *inv t* ∧ 1 ≤ *size t*) (*size going_to at_top within* {*t*. *inv t*})
using *eventually_going_to_at_top_linorder* **by** *force*
then show *?thesis*
unfolding *tree_filter_def* **.**
qed

The argument could already proceed here, but there is one pitfall. Nothing so far guarantees that the regions A_N are non-empty. Suppose some A_N were empty. Then *every* predicate, including λ*t*. *False*, would hold on all elements of A_N, as there are none. Since *eventually* only requires one witness region, every statement whatsoever would then hold eventually and the filter

collapses to the degenerate filter *bot*. Every asymptotic statement over it becomes vacuously true. Note that such statements would still be provable, they would merely carry no information. The following lemma constructs, for every N, an invariant tree of size exactly N, hence a member of A_N.

lemma *exists_inv_tree_of_size*: $\exists\, t.\ inv\ t \wedge size\ t = N$
proof (*induction N*)
  case *0*
  show *?case*
    using *inv_Leaf* **by** *force*
next
  case (*Suc N*)
  then obtain *t* **where** *t*: *inv t size t* = *N*
    by *blast*
  from *t* **have** *inv* (*join t a Leaf*) *size* (*join t a Leaf*) = *Suc N*
    by (*auto simp*: *join_size inv_Leaf intro*: *inv_join*)
  then show *?case*
    by *blast*
qed

Properness is now immediate.

lemma *tree_filter_proper*: *tree_filter* $\neq$ *bot*
proof
  assume *tree_filter* = *bot*
  then have *eventually* ($\lambda t.$ *False*) *tree_filter*
    by (*simp add*: *eventually_False*)
  then obtain *C* **where** *C*: $\forall\, t \in \{t.\ inv\ t\}.\ C \leq size\ t \longrightarrow False$
    unfolding *tree_filter_def eventually_going_to_at_top_linorder* **by** *blast*
  obtain *t* :: (*'a* × *'b*) *tree* **where** *inv t size t* = *C*
    using *exists_inv_tree_of_size* **by** *blast*
  with *C* **show** *False*
    by *auto*
qed

Because the work functions operate on pairs of trees, the product filter is used to extend *tree_filter* to a filter on pairs. As the bound is symmetric in the two sizes, no restriction on their order is needed.

definition *tree_pair_filter* :: (((*'a* × *'b*) *tree*) × (*'a* × *'b*) *tree*) *filter* **where**
  tree_pair_filter = *tree_filter* $\times_F$ *tree_filter*

corollary *tree_pair_filter_proper*: *tree_pair_filter* $\neq$ *bot*
  by (*simp add*: *tree_pair_filter_def prod_filter_eq_bot tree_filter_proper*)

corollary *eventually_tree_pair_filter*:
eventually ($\lambda x.$ ($\lambda t.$ *inv t* $\wedge$ *1* $\leq$ *size t*) (*fst x*) $\wedge$ ($\lambda t.$ *inv t* $\wedge$ *1* $\leq$ *size t*) (*snd x*))
  (*tree_filter* $\times_F$ *tree_filter*)
  using *eventually_prodI*[*OF eventually_tree_filter eventually_tree_filter*] **by** *simp*

Finally, the Big-O bound can be stated.

theorem *W_bound_bigo_generic*:
fixes *W jw* :: (*'a* × *'b*) *tree* ⇒ (*'a* × *'b*) *tree* ⇒ *real*
and *k c* :: *real*
assumes *W_nonneg*: ⋀*ta tb*. *0* ≤ *W ta tb*
assumes *decomp*: ⋀*ta tb*. ⟦*inv ta*; *inv tb*⟧
⟹ *W ta tb* ≤ *1* + *length* (*split_parts ta tb*) + *jw ta tb* + *split_work ta tb*
assumes *jw_piv*: ⋀*ta tb*. ⟦*inv ta*; *inv tb*⟧
⟹ *jw ta tb* ≤ (∑ *t* ← *split_parts ta tb*. *k* + *c* ∗ *rank t*)
assumes *k_pos*: *0* < *k* **and** *c_pos*: *0* < *c*
shows (λ(*ta*, *tb*). *W ta tb*) ∈ *O*[*tree_pair_filter*](λ(*ta*, *tb*).
size_min ta tb ∗ *log 2* (*size_max ta tb* / *size_min ta tb* + *1*))
proof −
let *?K* = *1* + *K_lin k c* + *K_log k c*
let *?W* = (λ(*ta*, *tb*). *W ta tb*)
let *?g* = (λ(*ta*, *tb*). *size_min ta tb* ∗ *log 2* (*size_max ta tb* / *size_min ta tb* + *1*))

— First, plug in the bound computed above
have *W ta tb* ≤ *?K* ∗ (*size_min ta tb* ∗ *log 2* (*size_max ta tb* / *size_min ta tb* + *1*))
if *inv ta inv tb* **for** *ta tb*
using *W_bound_absorb*[*OF that decomp*[*OF that*] *jw_piv*[*OF that*] *k_pos c_pos*] **.**

— The right-hand side is non-negative; together with *W_nonneg* this allows the norms in the Big-O form to be introduced later on
moreover have *nonneg*: *0* ≤ *size_min t u* ∗ *log 2* (*size_max t u* / *size_min t u* + *1*)
for *t u* :: (*'a* × *'b*) *tree*
by (*simp add*: *log_def*)

— On the product filter both components are eventually invariant and of size at least one (the witness region is $A_1 \times A_1$), which supplies every assumption of the pointwise bound.
ultimately have *eventually* (λ(*ta*, *tb*).
W ta tb ≤ *?K* ∗ (*size_min ta tb* ∗ *log 2* (*size_max ta tb* / *size_min ta tb* + *1*)))
tree_pair_filter
unfolding *tree_pair_filter_def*
using *eventually_tree_pair_filter* **by** (*auto elim*: *eventually_mono*)

— Since both sides of the inequality are non-negative, the norms demanded by the Big-O definition can be introduced
then have *eventually* (λ*x*. *norm* (*?W x*) ≤ *?K* ∗ *norm* (*?g x*)) *tree_pair_filter*
by (*auto simp*: *W_nonneg nonneg elim*: *eventually_mono*)

— Which can be directly restated in the form required for Big-O.
then have ∃ *b* > *0*. *eventually* (λ*x*. *norm* (*?W x*) ≤ *b* ∗ *norm* (*?g x*))

```
tree_pair_filter
    using K_lin_log_pos k_pos c_pos less_imp_le by meson

  then show ?thesis
    by blast
qed
```

Which also begets the canonical form phrased in terms of the natural logarithm.

```
corollary W_bound_bigo_generic_ln:
  fixes W jw :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ real
    and k c :: real
  assumes W_nonneg: ⋀ta tb. 0 ≤ W ta tb
  assumes decomp: ⋀ta tb. ⟦inv ta; inv tb⟧
      ⟹ W ta tb ≤ 1 + length (split_parts ta tb) + jw ta tb + split_work ta tb
  assumes jw_piv: ⋀ta tb. ⟦inv ta; inv tb⟧
      ⟹ jw ta tb ≤ (∑ t ← split_parts ta tb. k + c * rank t)
  assumes k_pos: 0 < k and c_pos: 0 < c
  shows (λ(ta, tb). W ta tb) ∈ O[tree_pair_filter](λ(ta, tb).
          size_min ta tb * ln (size_max ta tb / size_min ta tb + 1))
proof −
  let ?f = λ(ta, tb). size_min ta tb * log 2 (size_max ta tb / size_min ta tb + 1)
  let ?g = λ(ta, tb). size_min ta tb * ln (size_max ta tb / size_min ta tb + 1)

  have ∗:?f = (λx. (1 / (ln 2)) * ?g x)
    using log_def by auto
  have O[tree_pair_filter](λx. (1 / (ln 2)) * ?g x) = O[tree_pair_filter](?g)
    by simp
  then show ?thesis
     using W_bound_bigo_generic[OF W_nonneg decomp jw_piv k_pos c_pos]
by(simp add: ∗)
qed
```

7 Union

Start by defining the work function and show it decomposes as split- and join-work.

```
fun W_union :: ('a * 'b) tree ⇒ ('a * 'b) tree ⇒ real where
W_union t1 t2 =
  (if t1 = Leaf then 1
   else if t2 = Leaf then 1
   else case t1 of Node l1 (a,_) r1 ⇒
     (let (l2, _, r2) = split a t2;
          l' = union l1 l2;
          r' = union r1 r2
      in 1
        + W_split t2 a
        + W_union l1 l2
```

```
      + W_union r1 r2
      + W_join l' a r'))

fun join_work_union :: ('a*'b)tree ⇒ ('a*'b)tree ⇒ real where
join_work_union t1 t2 =
  (if t1 = Leaf then 0 else
   if t2 = Leaf then 0 else
   case t1 of Node l1 (a, _) r1 ⇒
   let (l2,_ ,r2) = split a t2;
       l' = union l1 l2; r' = union r1 r2
   in 1 + W_join l' a r' + join_work_union l1 l2 + join_work_union r1 r2)

declare W_union.simps[simp del]
declare join_work_union.simps[simp del]

lemma W_union_work_bound:
  assumes inv t1 inv t2
  shows W_union t1 t2
      ≤ 1 + length (split_parts t1 t2) + join_work_union t1 t2 + split_work t1
t2
  using assms
proof (induction t1 t2 rule: W_union.induct)
  case (1 t1 t2)
  then show ?case
    by (fastforce simp: W_union.simps[of t1 t2] split_work.simps[of t1 t2]
                        join_work_union.simps[of t1 t2] split_parts.simps[of t1 t2]
                        inv_union split_inv
                  split: prod.splits tree.splits
                  dest: split_inv)
qed
```

7.1 Bounding join-work

To instantiate the generic work bound, the join-work needs to fit the cost model. First, show that *union* produces results whose rank is bounded by the ranks of the input trees. The upper bound is a consequence of *rule_sub*. The lower bound combines balance, monotonicity and *split_rank_min*.

```
lemma union_rank_upper: ⟦inv t1; inv t2⟧ ⟹ rank (union t1 t2) ≤ rank t1 +
rank t2
proof (induction t1 t2 rule: union.induct)
  case (1 t1 t2)
  consider (leaves) t1 = Leaf ∨ t2 = Leaf | (nodes) t1 ≠ Leaf ∧ t2 ≠ Leaf by
auto
  then show ?case
  proof cases
    case leaves
    then show ?thesis
      using 1.prems by (auto simp: union.simps[of t1 t2] rule_empty)
  next
```

```
    case nodes
    then obtain l1 a b r1 where t1_def: t1 = Node l1 (a, b) r1
      by (cases t1) auto
    obtain l2 f r2 where SPL: split a t2 = (l2, f, r2)
      by (cases split a t2) auto
    have inv_sub: inv l1 inv r1 inv l2 inv r2
      using 1.prems t1_def SPL by auto
    let ?tl = union l1 l2 and ?tr = union r1 r2

    have res: union t1 t2 = join ?tl a ?tr
      using nodes by (simp add: t1_def SPL union.simps[of Node l1 (a, b) r1 t2])
    have IHl: rank ?tl ≤ rank l1 + rank l2 and IHr: rank ?tr ≤ rank r1 + rank
r2
      using 1 nodes t1_def SPL inv_sub by simp_all
    have rank ?tl ≤ rank l1 + rank t2 rank ?tr ≤ rank r1 + rank t2
      using IHl IHr split_rank[OF SPL 1.prems(2)] by linarith+
    then have rank (join ?tl a ?tr) ≤ rank t1 + rank t2
      using rule_sub[of ?tl l1 rank t2 ?tr r1 a b] inv_union inv_sub 1.prems(1)
      by (simp add: t1_def)
    then show ?thesis
      using res by simp
  qed
qed

lemma union_rank_lower: ⟦inv t1; inv t2⟧ ⟹ rank t1 ≤ rank (union t1 t2) +
(c_u / c_l) * rank t2
proof (induction t1 t2 rule: union.induct)
  case (1 t1 t2)
  have kappa: 0 ≤ c_u / c_l
    using c_vals by simp
  consider (leaves) t1 = Leaf ∨ t2 = Leaf | (nodes) t1 ≠ Leaf ∧ t2 ≠ Leaf by
auto
  then show ?case
  proof cases
    case leaves
    have nn: 0 ≤ (c_u / c_l) * rank t2
      using kappa rank_pos'[OF 1.prems(2)] mult_nonneg_nonneg by force
    show ?thesis
    using nn rank_pos'[OF 1.prems(2)] rule_empty leaves
    by (cases t1 = Leaf) (auto simp: union.simps)
  next
    case nodes
    then obtain l1 a b r1 where t1_def: t1 = Node l1 (a, b) r1
      by (cases t1) auto
    obtain l2 f r2 where SPL: split a t2 = (l2, f, r2)
      by (cases split a t2) auto
    have inv_sub: inv l1 inv r1 inv l2 inv r2
      using 1.prems t1_def SPL by auto
    let ?k = c_u / c_l
```

```
    let ?tl = union l1 l2 and ?tr = union r1 r2

    have IHl: rank l1 ≤ rank ?tl + ?k * rank l2 and IHr: rank r1 ≤ rank ?tr +
?k * rank r2
      using 1 nodes t1_def SPL inv_sub by auto
    have mono: max (rank ?tl) (rank ?tr) ≤ rank (join ?tl a ?tr)
      using rule_mono inv_union inv_sub by blast
    have bal: rank t1 ≤ min (rank l1) (rank r1) + c_u
      using inv_children_explD(3,4)[of l1 (a,b) r1] 1.prems(1) t1_def by auto
    have pieces: min (rank l2) (rank r2) ≤ rank t2 − c_l
      using split_rank_min[OF 1.prems(2) _ SPL] nodes by simp

    have rank t1 ≤ min (rank ?tl + ?k * rank l2) (rank ?tr + ?k * rank r2) + c_u
      using bal IHl IHr min.mono by fastforce
    also have ... ≤ max (rank ?tl) (rank ?tr) + ?k * min (rank l2) (rank r2) +
c_u
    by (cases rank l2 ≤ rank r2) fastforce+
    also have ... ≤ rank (join ?tl a ?tr) + ?k * (rank t2 − c_l) + c_u
      using mono mult_left_mono[OF pieces kappa] by linarith
    also have ... = rank (join ?tl a ?tr) + ?k * rank t2
      using c_vals by (simp add: field_simps)
    finally show ?thesis
      using nodes by (simp add: t1_def SPL union.simps[of Node l1 (a, b) r1 t2])
  qed
qed
```

With the two results above, the join-work of every step is bounded by a constant plus a constant multiple of the rank of the tree being split. The constants are named after the generic bound they feed into.

```
abbreviation KJ_union ≡ (1 + c_δ)
abbreviation KJc_union ≡ (1 + c_u / c_l)

lemma KJ_union_pos: 0 < 1 + c_δ
  using c_δ_def c_vals by linarith

lemma KJc_union_pos: 0 < 1 + c_u / c_l
  using c_vals by (simp add: add_pos_nonneg)

fun join_work_union' :: ('a*'b)tree ⇒ ('a*'b)tree ⇒ real where
join_work_union' t1 t2 =
  (if t1 = Leaf then 0 else
   if t2 = Leaf then 0 else
   case t1 of Node l1 (a, _) r1 ⇒
   let (l2,_ ,r2) = split a t2
      in 1 + c_δ + KJc_union * rank t2 + join_work_union' l1 l2 +
join_work_union' r1 r2)

lemma join_work_bounded:
  assumes inv t1 inv t2
```

shows *join_work_union t1 t2* $\leq$ *join_work_union′ t1 t2*
using *assms*
proof (*induction t1 t2 rule*: *join_work_union.induct*)
case (*1 t1 t2*)
show *?case*
proof (*cases t1 = Leaf* $\vee$ *t2 = Leaf*)
case *True*
then show *?thesis* **by** (*auto simp*: *join_work_union.simps*)
next
case *nodes*: *False*
then obtain *l1 x b r1* **where** *t1_def*: *t1 = Node l1* (*x*, *b*) *r1*
by (*cases t1*) *auto*
obtain *l2 b′ r2* **where** *split_def*: *split x t2* = (*l2*, *b′*, *r2*)
by (*cases split x t2*) *auto*
have *inv_sub*: *inv l1 inv r1 inv l2 inv r2*
using *1.prems t1_def split_def* **by** *auto*

let *?l* = *union l1 l2*
let *?r* = *union r1 r2*

have *IH*: *join_work_union l1 l2* $\leq$ *join_work_union′ l1 l2*
join_work_union r1 r2 $\leq$ *join_work_union′ r1 r2*
using *1 split_inv* **by** (*metis inv_Node local.split_def nodes t1_def*)+

have *Wj*: *W_join ?l x ?r* $\leq c_\delta$ + *KJc_union* $*$ *rank t2*
proof −
have *up*: *rank ?l* $\leq$ *rank l1* + *rank l2 rank ?r* $\leq$ *rank r1* + *rank r2*
using *union_rank_upper inv_sub* **by** *blast*+
have *lo*: *rank l1* $\leq$ *rank ?l* + (c_u / c_l) $*$ *rank l2*
rank r1 $\leq$ *rank ?r* + (c_u / c_l) $*$ *rank r2*
using *union_rank_lower inv_sub* **by** *blast*+
have *pieces*: *rank l2* $\leq$ *rank t2 rank r2* $\leq$ *rank t2*
using *split_rank*[*OF split_def 1.prems*(*2*)] **by** *simp_all*
then have (c_u / c_l) $*$ *rank l2* $\leq$ (c_u / c_l) $*$ *rank t2*
(c_u / c_l) $*$ *rank r2* $\leq$ (c_u / c_l) $*$ *rank t2*
using *c_vals* **by** (*intro mult_left_mono*; *simp*)+
moreover have |*rank l1* − *rank r1*| $\leq c_\delta$
using *dmax′ t1_def 1.prems*(*1*) **by** *blast*
ultimately show *?thesis*
using *up lo pieces* **unfolding** *W_join_def* **by** (*simp add*: *abs_le_iff algebra_simps*)
qed

have *join_work_union t1 t2 = 1 + W_join ?l x ?r + join_work_union l1 l2 + join_work_union r1 r2*
using *t1_def split_def nodes* **by** (*auto simp*: *join_work_union.simps split.simps split*: *prod.splits if_split*)
also have ... $\leq$ *1* + c_δ + *KJc_union* $*$ *rank t2* + *join_work_union′ l1 l2* + *join_work_union′ r1 r2*

```
    using Wj IH by linarith
  also have ... = join_work_union' t1 t2
    using t1_def split_def nodes by (auto simp: split.simps split: prod.splits
if_split)
  finally show ?thesis .
 qed
qed

lemma join_work_union'_as_sum:
 join_work_union' t1 t2 = (∑ t ← split_parts t1 t2. KJ_union + KJc_union *
rank t)
proof (induction t1 t2 rule: split_parts.induct)
 case (1 t1 t2)
 then show ?case
  by (force simp: join_work_union'.simps[of t1 t2] split_parts.simps[of t1 t2]
           split: tree.splits prod.split)
qed

corollary join_work_union_le_sum_rank:
 assumes inv t1 inv t2
  shows join_work_union t1 t2 ≤ (∑ t ← split_parts t1 t2. KJ_union +
KJc_union * rank t)
 using join_work_bounded[OF assms] join_work_union'_as_sum by auto
```

7.2 Final bound

The step of the set operations charges its unit of work against the final join, absorbing the constants of *rule_cost_W* into *K_T*. This form is shared by the union and intersection bridges.

```
lemma rule_cost_K_T:
 assumes inv l inv r
 shows 1 + real (T_join l a r) ≤ K_T + K_T * W_join l a r
proof −
 have 1 + real (T_join l a r) ≤ (1 + k0) + k1 * W_join l a r
  using rule_cost_W[OF assms] by fastforce
 also have ... ≤ K_T + K_T * W_join l a r
  using K_T_ge_1_plus_k0 K_T_ge_k1 by (intro add_mono mult_right_mono)
auto
 finally show ?thesis .
qed
```

Writing out the *union* specific induction step enables complete automation.

```
lemma rule_cost_step_union:
 assumes inv l inv r
    and real t_s ≤ K_T * w_s real t_l ≤ K_T * w_l real t_r ≤ K_T * w_r
 shows 1 + (real t_s + (real t_l + (real t_r + real (T_join l a r))))
       ≤ K_T * (1 + w_s + w_l + w_r + W_join l a r)
proof −
```

```
  have K_T * (1 + w_s + w_l + w_r + W_join l a r)
      = K_T * w_s + K_T * w_l + K_T * w_r + (K_T + K_T * W_join l a r)
    by (simp add: algebra_simps)
  then show ?thesis
    using assms(3-5) rule_cost_K_T[OF assms(1,2), where a=a] by linarith
qed

time_fun union equations union.simps
declare T_union.simps[simp del]

lemma T_union_bridge:
  ⟦inv t1; inv t2⟧ ⟹ real (T_union t1 t2) ≤ K_T * W_union t1 t2
proof (induction t1 t2 rule: union.induct)
  case (1 t1 t2)
  then show ?case
    using K_T_ge_1
   by (fastforce simp: T_union.simps[of t1 t2] W_union.simps[of t1 t2] inv_union
split_inv
                  intro!: rule_cost_step_union T_split_bridge
                  split: tree.splits prod.splits)
qed
```

The generic work bounds can be instantiated directly.

```
theorem W_union_bound:
  assumes inv t1 inv t2
  shows W_union t1 t2
      ≤ 1 + K_lin KJ_union KJc_union * size_min t1 t2
        + K_log KJ_union KJc_union * (size_min t1 t2 * log 2 (size_max t1 t2
/ size_min t1 t2 + 1))
proof −
  have jw: join_work_union t1 t2 ≤ (∑ t ← split_parts t1 t2. KJ_union +
KJc_union * rank t)
    using join_work_union_le_sum_rank[OF assms] by simp
  show ?thesis
     using W_bound_generic'[OF assms W_union_work_bound[OF assms] jw
KJ_union_pos KJc_union_pos]
    by simp
qed

corollary T_union_bound:
  assumes inv t1 inv t2
  shows real (T_union t1 t2)
      ≤ K_T + (K_T * K_lin KJ_union KJc_union) * size_min t1 t2
          + (K_T * K_log KJ_union KJc_union) * (size_min t1 t2 * log 2
(size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have real (T_union t1 t2) ≤ K_T * W_union t1 t2
    using T_union_bridge assms(1,2) by simp
  also have ... ≤ K_T * (1 + K_lin KJ_union KJc_union * size_min t1 t2 +
```

```
    K_log KJ_union KJc_union * (size_min t1 t2 * log 2 (size_max t1 t2 /
size_min t1 t2 + 1)))
   using W_union_bound[OF assms] K_T_ge_1 by auto
 also have ... = K_T + (K_T * K_lin KJ_union KJc_union) * size_min t1
t2 +
   (K_T * K_log KJ_union KJc_union) * (size_min t1 t2 * log 2 (size_max t1
t2 / size_min t1 t2 + 1))
   by argo
 finally show ?thesis .
qed
```

The generic asymptotic bound requires the work functions to be non-negative.

```
lemma W_split_nonneg: 0 ≤ W_split t x
 by (induction t x rule: W_split.induct)
    (auto split: cmp_val.split prod.split)

lemma W_union_nonneg: 0 ≤ W_union t1 t2
proof (induction t1 t2 rule: W_union.induct)
 case (1 t1 t2)
 then show ?case
   by (auto simp: W_union.simps[of t1 t2]
         intro!: add_nonneg_nonneg W_split_nonneg
         split: tree.splits prod.splits)
qed

theorem W_union_bigo:
 (λ(t1, t2). W_union t1 t2) ∈ O[tree_pair_filter](λ(t1,t2).
   size_min t1 t2 * ln (size_max t1 t2 / size_min t1 t2 + 1))
proof −
 have ⋀ta tb. 0 ≤ W_union ta tb
   by (rule W_union_nonneg)
 moreover have ⋀ta tb. ⟦inv ta; inv tb⟧ ⟹ W_union ta tb
    ≤ 1 + length (split_parts ta tb) + join_work_union ta tb + split_work ta tb
   using W_union_work_bound by blast
 moreover have ⋀ta tb. ⟦inv ta; inv tb⟧
      ⟹ join_work_union ta tb ≤ (∑ t ← split_parts ta tb. KJ_union +
KJc_union * rank t)
   using join_work_union_le_sum_rank by fastforce
 ultimately show ?thesis
    using KJ_union_pos KJc_union_pos W_bound_bigo_generic_ln[where
k=KJ_union and c=KJc_union] by force
qed
```

The time bound follows from transitivity of Big-O.

```
corollary T_union_bigo:
 (λ(t1, t2). real (T_union t1 t2)) ∈ O[tree_pair_filter](λ(t1,t2).
   size_min t1 t2 * ln (size_max t1 t2 / size_min t1 t2 + 1))
proof −
```

```
  have eventually (λx. norm ((λ(t1, t2). real (T_union t1 t2)) x)
        ≤ K_T * norm ((λ(t1, t2). W_union t1 t2) x)) tree_pair_filter
    unfolding tree_pair_filter_def using eventually_tree_pair_filter
        by (auto simp: W_union_nonneg elim!: eventually_mono intro!:
T_union_bridge)
  then have (λ(t1, t2). real (T_union t1 t2))
     ∈ O[tree_pair_filter](λ(t1, t2). W_union t1 t2)
    using bigoI by fast
  then show ?thesis
    using W_union_bigo landau_o.big_trans by blast
qed
```

8 Intersection

Start by defining the work function and show it decomposes into the split- and join-work.

```
fun W_inter :: ('a * 'b) tree ⇒ ('a * 'b) tree ⇒ real where
W_inter t1 t2 =
  (if t1 = Leaf then 1
   else if t2 = Leaf then 1
   else case t1 of Node l1 (a,_) r1 ⇒
     (let (l2, b, r2) = split a t2;
          l' = inter l1 l2;
          r' = inter r1 r2
      in 1
        + W_split t2 a
        + (if b then W_join l' a r' else W_join2 l' r')
        + W_inter l1 l2
        + W_inter r1 r2
        ))

fun join_work_inter :: ('a*'b)tree ⇒ ('a*'b)tree ⇒ real where
join_work_inter t1 t2 =
  (if t1 = Leaf then 0 else
   if t2 = Leaf then 0 else
   case t1 of Node l1 (a, _) r1 ⇒
   let (l2,b ,r2) = split a t2;
       l' = inter l1 l2; r' = inter r1 r2
   in 1 + (if b then W_join l' a r' else W_join2 l' r') + join_work_inter l1 l2 +
join_work_inter r1 r2)

declare W_inter.simps[simp del]
declare join_work_inter.simps[simp del]

lemma W_inter_work_bound:
  assumes inv t1 inv t2
  shows W_inter t1 t2
      ≤ 1 + length (split_parts t1 t2) + join_work_inter t1 t2 + split_work t1 t2
```

```
  using assms
proof (induction t1 t2 rule: W_inter.induct)
  case (1 t1 t2)
  then show ?case
    by (fastforce simp: W_inter.simps[of t1 t2] split_work.simps[of t1 t2]
                        join_work_inter.simps[of t1 t2] split_parts.simps[of t1 t2]
                        inv_inter split_inv
                  split: prod.splits tree.splits
                  dest: split_inv)
qed
```

8.1 Bounding join-work

The rank-based argument for *local.union* does not carry over, as there is no submodularity rule for *join2*. By nature of the operation, however, the size of the resulting tree can be bounded directly.

```
lemma inter_size_bound:
  assumes inv t1 inv t2
  shows size (inter t1 t2) ≤ size t2
using assms
proof (induction t1 t2 rule: inter.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t1 = Leaf ∨ t2 = Leaf)
    case True
    then show ?thesis using inter.simps by auto
  next
    case False
    then obtain l1 a b r1 where t1_def: t1 = Node l1 (a, b) r1
      by (cases t1) auto
    obtain l2 f r2 where SPL: split a t2 = (l2, f, r2)
      by (cases split a t2) auto
    have invs: inv l1 inv r1 inv l2 inv r2
      using 1.prems t1_def SPL inv_Node split_inv by metis+
    have IHl: size (inter l1 l2) ≤ size l2 and IHr: size (inter r1 r2) ≤ size r2
      using 1.IH False t1_def SPL invs by auto
    have result: inter t1 t2 = (if f then join (inter l1 l2) a (inter r1 r2)
                                     else join2 (inter l1 l2) (inter r1 r2))
     using False t1_def SPL by (auto simp: inter.simps split: prod.splits tree.splits)
    show ?thesis
    proof (cases f)
      case True
      then have size (inter t1 t2) = size (inter l1 l2) + size (inter r1 r2) + 1
        using result join_size inv_inter invs by auto
      moreover have size l2 + size r2 = size t2 − 1
        using split_size_true SPL True 1.prems(2) by (metis (full_types))
      moreover have 0 < size t2
        using False by (cases t2) auto
      ultimately show ?thesis
```

```
      using IHl IHr by linarith
    next
      case FalseF: False
      then have size (inter t1 t2) = size (inter l1 l2) + size (inter r1 r2)
        using result join2_size inv_inter invs by auto
      then show ?thesis
        using IHl IHr split_size[OF SPL[symmetric] 1.prems(2)] by linarith
    qed
  qed
qed
```

By balance, *size* can be exchanged for *rank* to arrive at a shape required for the generic work bound. This pattern is re-used by difference.

```
lemma log_size1_le_rank_ratio:
  assumes inv t
  shows c_u * log 2 (size1 t) ≤ (c_u / c_l) * rank t + c_u
proof −
  have real (size1 t) ≤ 2 ^ height t
    using size1_height by auto
  then have log 2 (real (size1 t)) ≤ log 2 (2 ^ height t)
    by (intro log_mono) auto
  then have log 2 (size1 t) ≤ height t
     by fastforce
  also have ... ≤ rank t / c_l + 1
    using rank_lower_height_inverse' assms by fastforce
  finally have c_u * log 2 (size1 t) ≤ c_u * (rank t / c_l + 1)
    using c_vals assms rank_lower_height_inverse' by auto
  then show ?thesis by argo
qed

corollary inter_rank_le_log_size:
  assumes inv t1 inv t2
  shows rank (inter t1 t2) ≤ c_u * log 2 (size1 t2)
proof −
  have inv (inter t1 t2)
    using assms inv_inter by auto
  then have rank (inter t1 t2) ≤ c_u * log 2 (size1 (inter t1 t2))
    using rank_le_cu_log_size1 by blast
  also have ... ≤ c_u * log 2 (size1 t2)
    using inter_size_bound[OF assms] c_vals by (simp add: size1_size)
  finally show ?thesis .
qed
```

The join work can thus be bounded.

```
fun join_work_inter' :: ('a*'b)tree ⇒ ('a*'b)tree ⇒ real where
join_work_inter' t1 t2 =
  (if t1 = Leaf then 0 else
   if t2 = Leaf then 0 else
   case t1 of Node l1 (a, _) r1 ⇒
```

```
  let (l2, _, r2) = split a t2;
      l' = inter l1 l2; r' = inter r1 r2
  in 1 + W_join l' a r' + 1 + c_u + K_min * rank r'
     + join_work_inter' l1 l2 + join_work_inter' r1 r2)

declare join_work_inter'.simps[simp del]

lemma join_work_inter_le_inter':
  assumes inv t1 inv t2
  shows join_work_inter t1 t2 ≤ join_work_inter' t1 t2
using assms
proof (induction t1 t2 rule: join_work_inter.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t1 = Leaf ∨ t2 = Leaf)
    case True
    then show ?thesis
      by (auto simp: join_work_inter.simps join_work_inter'.simps)
  next
    case False
    then obtain l1 a b r1 where t1_def: t1 = Node l1 (a, b) r1
      by (cases t1) auto
    obtain l2 f r2 where SPL: split a t2 = (l2, f, r2)
      by (cases split a t2) auto
    have inv_l1: inv l1 and inv_r1: inv r1
      using 1.prems(1) t1_def inv_Node by blast+
    have inv_l2: inv l2 and inv_r2: inv r2
      using SPL 1.prems(2) split_inv by metis+

    have W_join2 (inter l1 l2) (inter r1 r2)
        ≤ W_join (inter l1 l2) a (inter r1 r2) + 1 + c_u + K_min * rank (inter
r1 r2)
       using W_join2_bounded[OF inv_inter[OF inv_l1 inv_l2] inv_inter[OF
inv_r1 inv_r2]] .
    moreover have join_work_inter l1 l2 ≤ join_work_inter' l1 l2
      using 1.IH False t1_def SPL inv_l1 inv_l2 by auto
    moreover have join_work_inter r1 r2 ≤ join_work_inter' r1 r2
      using 1.IH False t1_def SPL inv_r1 inv_r2 by auto
    moreover have K_min * rank (inter r1 r2) ≥ 0
     using K_min_def c1_pos c_vals rank_pos'[OF inv_inter[OF inv_r1 inv_r2]]
      by (auto intro: mult_nonneg_nonneg divide_nonneg_pos)
    ultimately  show ?thesis using False t1_def SPL c_vals
      by (auto simp: join_work_inter.simps[of Node l1 (a,b) r1 t2]
                     join_work_inter'.simps[of Node l1 (a,b) r1 t2])
  qed
qed

lemma cu_log_size1_mono:
  fixes s t :: ('a * 'b) tree
```

```
  assumes size s ≤ size t
  shows c_u * log 2 (size1 s) ≤ c_u * log 2 (size1 t)
  using assms c_vals by (auto simp: size1_size intro: mult_left_mono log_mono)

lemma join_work_inter'_as_sum:
  assumes inv t1 inv t2
  shows join_work_inter' t1 t2
      ≤ (∑ t ← split_parts t1 t2.
          2 + c_u + (2 + K_min) * c_u * log 2 (size1 t))
using assms
proof (induction t1 t2 rule: split_parts.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t1 = Leaf ∨ t2 = Leaf)
    case True
    then show ?thesis
      by (auto simp: join_work_inter'.simps split_parts.simps)
  next
    case False
    then obtain l1 a b r1 where t1_def: t1 = Node l1 (a, b) r1
      by (cases t1) auto
    obtain l2 f r2 where SPL: split a t2 = (l2, f, r2)
      by (cases split a t2) auto
    have inv_l1: inv l1 and inv_r1: inv r1 and inv_l2: inv l2 and inv_r2: inv
r2
      using 1.prems t1_def SPL inv_Node split_inv by metis+
    have inv_il: inv (inter l1 l2) and inv_ir: inv (inter r1 r2)
      using inv_inter inv_l1 inv_r1 inv_l2 inv_r2 by auto
    have sz: size l2 ≤ size t2 size r2 ≤ size t2
      using split_size[OF SPL[symmetric] 1.prems(2)] by linarith+
    have rank_il: rank (inter l1 l2) ≤ c_u * log 2 (size1 t2)
     using inter_rank_le_log_size[OF inv_l1 inv_l2] cu_log_size1_mono sz by
force
    have rank_ir: rank (inter r1 r2) ≤ c_u * log 2 (size1 t2)
     using inter_rank_le_log_size[OF inv_r1 inv_r2] cu_log_size1_mono sz by
force

    have W_join (inter l1 l2) a (inter r1 r2) ≤ 2 * c_u * log 2 (size1 t2)
       using rank_il rank_ir W_join_def rank_pos'[OF inv_il] rank_pos'[OF
inv_ir]
      by (simp add: abs_le_iff)
    then have 1 + W_join (inter l1 l2) a (inter r1 r2) + 1 + c_u + K_min *
rank (inter r1 r2)
                ≤ 2 + c_u + (2 + K_min) * c_u * log 2 (size1 t2)
      using mult_left_mono[OF rank_ir K_min_nonneg] by (simp add: alge-
bra_simps)
    moreover have split_parts t1 t2 = t2 # (split_parts l1 l2 @ split_parts r1
r2)
      using False t1_def SPL by (simp add: split_parts.simps)
```

```
    moreover have join_work_inter' t1 t2
        = 1 + W_join (inter l1 l2) a (inter r1 r2) + 1 + c_u + K_min * rank
(inter r1 r2)
          + join_work_inter' l1 l2 + join_work_inter' r1 r2
      using False t1_def SPL
      by (auto simp: join_work_inter'.simps split: prod.splits)
    moreover have join_work_inter' l1 l2
          ≤ (∑ t ← split_parts l1 l2. 2 + c_u + (2 + K_min) * c_u * log 2 (size1
t))
     and join_work_inter' r1 r2
          ≤ (∑ t ← split_parts r1 r2. 2 + c_u + (2 + K_min) * c_u * log 2 (size1
t))
      using 1.IH False t1_def SPL inv_l1 inv_r1 inv_l2 inv_r2 by auto
    ultimately show ?thesis
      by simp
  qed
qed
```

Exchanging the log for the rank via *log_size1_le_rank_ratio* puts the sum directly into the shape $k + c * rank\ t$ accepted by the generic bound.

```
abbreviation KJk_inter ≡ 2 + c_u + (2 + K_min) * c_u
abbreviation KJc_inter ≡ (2 + K_min) * c_u / c_l

lemma join_work_inter_le_sum_rank:
  assumes inv t1 inv t2
  shows join_work_inter t1 t2
      ≤ (∑ t ← split_parts t1 t2. KJk_inter + KJc_inter * rank t)
proof −
  have pw: 2 + c_u + (2 + K_min) * c_u * log 2 (size1 t)
        ≤ KJk_inter + KJc_inter * rank t
    if t ∈ set (split_parts t1 t2) for t
  proof −
    have inv_t: inv t using split_parts_inv[OF assms(2)] that by simp
    have (2 + K_min) * (c_u * log 2 (size1 t)) ≤ (2 + K_min) * ((c_u / c_l) * rank
t + c_u)
      using log_size1_le_rank_ratio[OF inv_t] K_min_nonneg by (auto intro:
mult_left_mono)
    then show ?thesis by argo
  qed

  have join_work_inter t1 t2 ≤ join_work_inter' t1 t2
    using join_work_inter_le_inter'[OF assms(1,2)] .
  also have ... ≤ (∑ t ← split_parts t1 t2. 2 + c_u + (2 + K_min) * c_u * log 2
(size1 t))
    using join_work_inter'_as_sum[OF assms(1,2)] .
  also have ... ≤ (∑ t ← split_parts t1 t2. KJk_inter + KJc_inter * rank t)
    using pw by (intro sum_list_mono) auto
  finally show ?thesis .
qed
```

8.2 Final bound

The remaining is analogous to the instantiation for *union*.

lemma *rule_cost_step_inter*:
  assumes *inv l inv r*
    and *real* $t_s \leq K\_T * w_s$ *real* $t_l \leq K\_T * w_l$ *real* $t_r \leq K\_T * w_r$
  shows $1 + (real\ t_s + (real\ t_l + (real\ t_r + real\ (T\_join\ l\ a\ r))))$
      $\leq K\_T * (1 + w_s + W\_join\ l\ a\ r + w_l + w_r)$
proof −
  have $K\_T * (1 + w_s + W\_join\ l\ a\ r + w_l + w_r)$
      $= K\_T * w_s + K\_T * w_l + K\_T * w_r + (K\_T + K\_T * W\_join\ l\ a\ r)$
    by (*simp add*: *algebra_simps*)
  then show *?thesis*
    using *assms*(3−5) *rule_cost_K_T*[*OF assms*(1,2), **where** *a*=*a*] **by** *linarith*
qed

lemma *rule_cost_step_inter2*:
  assumes *inv l inv r*
    and *real* $t_s \leq K\_T * w_s$ *real* $t_l \leq K\_T * w_l$ *real* $t_r \leq K\_T * w_r$
  shows $1 + (real\ t_s + (real\ t_l + (real\ t_r + real\ (T\_join2\ l\ r))))$
      $\leq K\_T * (1 + w_s + W\_join2\ l\ r + w_l + w_r)$
proof −
  have $K\_T * (1 + w_s + W\_join2\ l\ r + w_l + w_r)$
      $= K\_T * w_s + K\_T * w_l + K\_T * w_r + (K\_T + K\_T * W\_join2\ l\ r)$
    by *argo*
  then show *?thesis*
    using *assms*(3−5) *T_join2_bridge*[*OF assms*(1,2)] *K_T_ge_1* **by** *linarith*
qed

time_fun *inter* **equations** *inter.simps*
declare *T_inter.simps*[*simp del*]

lemma *T_inter_bridge*:
  ⟦*inv t1*; *inv t2*⟧ ⟹ *real* (*T_inter t1 t2*) ≤ *K_T* * *W_inter t1 t2*
proof (*induction t1 t2 rule*: *inter.induct*)
  case (*1 t1 t2*)
  then show *?case*
    using *K_T_ge_1*
    by (*fastforce simp*: *T_inter.simps*[*of t1 t2*] *W_inter.simps*[*of t1 t2*] *inv_inter split_inv*
        simp del: *T_join2.simps*
        intro!: *rule_cost_step_inter rule_cost_step_inter2 T_split_bridge*
        split: *tree.splits prod.splits*)

qed

lemma *KJk_inter_pos*: *0* < *KJk_inter*
proof −
  have $0 \leq (2 + K\_min) * c_u$

```
  using K_min_nonneg c_u_nonneg by (auto intro: mult_nonneg_nonneg)
 then show ?thesis using c_u_ge_one by linarith
qed

lemma KJc_inter_pos: 0 < KJc_inter
 using K_min_nonneg c_vals by auto

theorem W_inter_bound:
 assumes inv t1 inv t2
 shows W_inter t1 t2
     ≤ 1 + K_lin KJk_inter KJc_inter * size_min t1 t2
       + K_log KJk_inter KJc_inter
         * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
 using W_bound_generic'[OF assms W_inter_work_bound[OF assms]
       join_work_inter_le_sum_rank[OF assms]
       KJk_inter_pos KJc_inter_pos] by simp

corollary T_inter_bound:
 assumes inv t1 inv t2
 shows real (T_inter t1 t2)
     ≤ K_T + (K_T * K_lin KJk_inter KJc_inter) * size_min t1 t2
       + (K_T * K_log KJk_inter KJc_inter)
         * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
proof −
 have real (T_inter t1 t2) ≤ K_T * W_inter t1 t2
   using T_inter_bridge assms(1,2) by simp
 also have ... ≤ K_T * (1 + K_lin KJk_inter KJc_inter * size_min t1 t2 +
     K_log KJk_inter KJc_inter * (size_min t1 t2 * log 2 (size_max t1 t2 /
size_min t1 t2 + 1)))
   using W_inter_bound[OF assms] K_T_ge_1 by auto
 also have ... = K_T + (K_T * K_lin KJk_inter KJc_inter) * size_min t1
t2 +
   (K_T * K_log KJk_inter KJc_inter)
     * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
   by argo
 finally show ?thesis .
qed

lemma W_split_min_nonneg: t ≠ Leaf ⟹ 0 ≤ W_split_min t
 by (induction t rule: W_split_min.induct)
    (auto intro!: add_nonneg_nonneg split: prod.split)

lemma W_join2_nonneg: 0 ≤ W_join2 l r
 unfolding W_join2_def
 by (auto intro!: add_nonneg_nonneg W_split_min_nonneg split: prod.split)

lemma W_inter_nonneg: 0 ≤ W_inter t1 t2
proof (induction t1 t2 rule: W_inter.induct)
 case (1 t1 t2)
```

```
  then show ?case
    by (auto simp: W_inter.simps[of t1 t2]
          intro!: add_nonneg_nonneg W_split_nonneg W_join2_nonneg
          split: tree.splits prod.splits)
qed


theorem W_inter_bigo:
 (λ(t1, t2). W_inter t1 t2) ∈ O[tree_pair_filter](λ(t1,t2).
   size_min t1 t2 * ln (size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have ⋀ta tb. 0 ≤ W_inter ta tb
    by (rule W_inter_nonneg)
  moreover have ⋀ta tb. ⟦inv ta; inv tb⟧ ⟹ W_inter ta tb
      ≤ 1 + length (split_parts ta tb) + join_work_inter ta tb + split_work ta tb
    using W_inter_work_bound by blast
  moreover have ⋀ta tb. ⟦inv ta; inv tb⟧
    ⟹ join_work_inter ta tb ≤ (∑ t ← split_parts ta tb. KJk_inter + KJc_inter
* rank t)
    using join_work_inter_le_sum_rank by blast
  ultimately show ?thesis
    using KJk_inter_pos KJc_inter_pos
          W_bound_bigo_generic_ln[where k=KJk_inter and c=KJc_inter] by
force
qed

corollary T_inter_bigo:
 (λ(t1, t2). real (T_inter t1 t2)) ∈ O[tree_pair_filter](λ(t1,t2).
   size_min t1 t2 * ln (size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have eventually (λx. norm ((λ(t1, t2). real (T_inter t1 t2)) x)
        ≤ K_T * norm ((λ(t1, t2). W_inter t1 t2) x)) tree_pair_filter
    unfolding tree_pair_filter_def using eventually_tree_pair_filter
   by (auto simp: W_inter_nonneg elim!: eventually_mono intro!: T_inter_bridge)
  then have (λ(t1, t2). real (T_inter t1 t2))
      ∈ O[tree_pair_filter](λ(t1, t2). W_inter t1 t2)
    using bigoI by fast
  then show ?thesis
    using W_inter_bigo landau_o.big_trans by blast
qed
```

9 Difference

Difference decomposes *t1* instead of *t2*. Otherwise the proof proceeds in an identical manner.

```
fun W_diff :: ('a * 'b) tree ⇒ ('a * 'b) tree ⇒ real where
W_diff t1 t2 =
  (if t1 = Leaf then 1
```

```
  else if t2 = Leaf then 1
  else case t2 of Node l2 (a,_) r2 ⇒
   (let (l1, _, r1) = split a t1;
        l' = diff l1 l2; r' = diff r1 r2
    in 1
       + W_split t1 a
       + W_join2 l' r'
       + W_diff l1 l2
       + W_diff r1 r2))

fun join_work_diff :: ('a*'b)tree ⇒ ('a*'b)tree ⇒ real where
join_work_diff t1 t2 =
  (if t1 = Leaf then 0 else
   if t2 = Leaf then 0 else
   case t2 of Node l2 (a, _) r2 ⇒
   let (l1, _, r1) = split a t1;
       l' = diff l1 l2; r' = diff r1 r2
   in 1 + W_join2 l' r' + join_work_diff l1 l2 + join_work_diff r1 r2)

declare W_diff.simps[simp del]
declare join_work_diff.simps[simp del]

lemma W_diff_work_bound:
  assumes inv t1 inv t2
  shows W_diff t1 t2
      ≤ 1 + length (split_parts t2 t1) + join_work_diff t1 t2 + split_work t2 t1
  using assms
proof (induction t1 t2 rule: W_diff.induct)
  case (1 t1 t2)
  then show ?case
    by (fastforce simp: W_diff.simps[of t1 t2] split_work.simps[of t2 t1]
                        join_work_diff.simps[of t1 t2] split_parts.simps[of t2 t1]
                        inv_diff split_inv
                  split: prod.splits tree.splits
                  dest: split_inv)
qed
```

9.1 Bounding join-work

```
lemma diff_size_bound:
  assumes inv t1 inv t2
  shows size (diff t1 t2) ≤ size t1
using assms
proof (induction t1 t2 rule: diff.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t1 = Leaf ∨ t2 = Leaf)
    case True
    then show ?thesis by (auto simp: diff.simps[of t1 t2])
```

```
  next
    case False
    then obtain l2 a b r2 where t2_def: t2 = Node l2 (a, b) r2
      by (cases t2) auto
    obtain l1 f r1 where SPL: split a t1 = (l1, f, r1)
      by (cases split a t1) auto
    have inv_l1: inv l1 and inv_r1: inv r1
      using SPL 1.prems(1) split_inv by metis+
    have inv_l2: inv l2 and inv_r2: inv r2
      using 1.prems(2) t2_def inv_Node by blast+
    have IHl: size (diff l1 l2) ≤ size l1
      using 1.IH(1) False t2_def SPL inv_l1 inv_l2 by auto
    have IHr: size (diff r1 r2) ≤ size r1
      using 1.IH(2) False t2_def SPL inv_r1 inv_r2 by auto

    have diff t1 t2 = join2 (diff l1 l2) (diff r1 r2)
     using False t2_def SPL by (auto simp: diff.simps split: prod.splits tree.splits)
    then have size (diff t1 t2) = size (diff l1 l2) + size (diff r1 r2)
     by (simp add: join2_size[OF inv_diff[OF inv_l1 inv_l2] inv_diff[OF inv_r1
inv_r2]])
    also have ... ≤ size l1 + size r1
      using IHl IHr by linarith
    also have ... ≤ size t1
      using split_size[OF SPL[symmetric] 1.prems(1)] .
    finally show ?thesis .
  qed
qed

lemma diff_rank_le_log_size:
  assumes inv t1 inv t2
  shows rank (diff t1 t2) ≤ c_u * log 2 (size1 t1)
proof −
  have inv (diff t1 t2) using assms inv_diff by auto
  then have rank (diff t1 t2) ≤ c_u * log 2 (size1 (diff t1 t2))
    using rank_le_cu_log_size1 by blast
  also have ... ≤ c_u * log 2 (size1 t1)
    using diff_size_bound[OF assms] c_vals by (simp add: size1_size)
  finally show ?thesis .
qed

fun join_work_diff' :: ('a*'b)tree ⇒ ('a*'b)tree ⇒ real where
join_work_diff' t1 t2 =
  (if t1 = Leaf then 0 else
   if t2 = Leaf then 0 else
   case t2 of Node l2 (a, _) r2 ⇒
   let (l1, _, r1) = split a t1;
       l' = diff l1 l2; r' = diff r1 r2
   in 1 + W_join l' a r' + 1 + c_u + K_min * rank r'
      + join_work_diff' l1 l2 + join_work_diff' r1 r2)
```

```
declare join_work_diff'.simps[simp del]

lemma join_work_diff_le_diff':
  assumes inv t1 inv t2
  shows join_work_diff t1 t2 ≤ join_work_diff' t1 t2
using assms
proof (induction t1 t2 rule: join_work_diff.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t1 = Leaf ∨ t2 = Leaf)
    case True then show ?thesis
      by (auto simp: join_work_diff.simps join_work_diff'.simps)
  next
    case False
    then obtain l2 a b r2 where t2_def: t2 = Node l2 (a, b) r2
      by (cases t2) auto
    obtain l1 f r1 where SPL: split a t1 = (l1, f, r1)
      by (cases split a t1) auto
    have inv_l1: inv l1 and inv_r1: inv r1
      using SPL 1.prems(1) split_inv by metis+
    have inv_l2: inv l2 and inv_r2: inv r2
      using 1.prems(2) t2_def inv_Node by blast+
    have inv_dl: inv (diff l1 l2) and inv_dr: inv (diff r1 r2)
      using inv_diff inv_l1 inv_r1 inv_l2 inv_r2 by auto

    have join_work_diff l1 l2 ≤ join_work_diff' l1 l2
      using 1.IH(1) False t2_def SPL inv_l1 inv_l2 by auto
    moreover have join_work_diff r1 r2 ≤ join_work_diff' r1 r2
      using 1.IH(2) False t2_def SPL inv_r1 inv_r2 by auto
    moreover have W_join2 (diff l1 l2) (diff r1 r2)
       ≤ W_join (diff l1 l2) a (diff r1 r2) + 1 + c_u + K_min * rank (diff r1 r2)
      using W_join2_bounded[OF inv_dl inv_dr] .
    ultimately show ?thesis
      using False t2_def SPL K_min_def c1_pos c_vals
      by (auto simp: join_work_diff.simps join_work_diff'.simps W_join_def
            split: prod.splits)
  qed
qed

lemma join_work_diff'_as_sum:
  assumes inv t1 inv t2
  shows join_work_diff' t1 t2
      ≤ (∑ t ← split_parts t2 t1.
          2 + c_u + (2 + K_min) * c_u * log 2 (size1 t))
using assms
proof (induction t1 t2 rule: diff.induct)
  case (1 t1 t2)
  show ?case
```

```
  proof (cases t1 = Leaf ∨ t2 = Leaf)
    case True then show ?thesis
      by (auto simp: join_work_diff'.simps split_parts.simps)
  next
    case False
    then obtain l2 a b r2 where t2_def: t2 = Node l2 (a, b) r2
      by (cases t2) auto
    obtain l1 f r1 where SPL: split a t1 = (l1, f, r1)
      by (cases split a t1) auto
    have inv_l1: inv l1 and inv_r1: inv r1 and inv_l2: inv l2 and inv_r2: inv
r2
      using 1.prems t2_def SPL inv_Node split_inv by metis+
    have inv_dl: inv (diff l1 l2) and inv_dr: inv (diff r1 r2)
      using inv_diff inv_l1 inv_r1 inv_l2 inv_r2 by auto
    have sz: size l1 ≤ size t1 size r1 ≤ size t1
      using split_size[OF SPL[symmetric] 1.prems(1)] by linarith+
    have rank_dl: rank (diff l1 l2) ≤ c_u * log 2 (size1 t1)
      using diff_rank_le_log_size[OF inv_l1 inv_l2] cu_log_size1_mono sz by
force
    have rank_dr: rank (diff r1 r2) ≤ c_u * log 2 (size1 t1)
      using diff_rank_le_log_size[OF inv_r1 inv_r2] cu_log_size1_mono sz by
force

    have W_join (diff l1 l2) a (diff r1 r2) ≤ 2 * c_u * log 2 (size1 t1)
      using rank_dl rank_dr W_join_def rank_pos'[OF inv_dl] rank_pos'[OF
inv_dr]
      by (simp add: abs_le_iff)
    then have 1 + W_join (diff l1 l2) a (diff r1 r2) + 1 + c_u + K_min * rank
(diff r1 r2)
                ≤ 2 + c_u + (2 + K_min) * c_u * log 2 (size1 t1)
      using mult_left_mono[OF rank_dr K_min_nonneg] by (simp add: alge-
bra_simps)
    moreover have split_parts t2 t1 = t1 # (split_parts l2 l1 @ split_parts r2
r1)
      using False t2_def SPL by (simp add: split_parts.simps)
    moreover have join_work_diff' t1 t2
        = 1 + W_join (diff l1 l2) a (diff r1 r2) + 1 + c_u + K_min * rank (diff
r1 r2)
          + join_work_diff' l1 l2 + join_work_diff' r1 r2
      using False t2_def SPL
      by (auto simp: join_work_diff'.simps split: prod.splits)
    moreover have join_work_diff' l1 l2
          ≤ (∑ t ← split_parts l2 l1. 2 + c_u + (2 + K_min) * c_u * log 2 (size1
t))
     and join_work_diff' r1 r2
          ≤ (∑ t ← split_parts r2 r1. 2 + c_u + (2 + K_min) * c_u * log 2 (size1
t))
      using 1.IH False t2_def SPL inv_l1 inv_r1 inv_l2 inv_r2 by auto
    ultimately show ?thesis
```

```
    by simp
  qed
qed

abbreviation KJk_diff ≡ 2 + c_u + (2 + K_min) * c_u
abbreviation KJc_diff ≡ (2 + K_min) * c_u / c_l

lemma join_work_diff_le_sum_rank:
  assumes inv t1 inv t2
  shows join_work_diff t1 t2
      ≤ (∑ t ← split_parts t2 t1. KJk_diff + KJc_diff * rank t)
proof −
  have pw: 2 + c_u + (2 + K_min) * c_u * log 2 (size1 t)
        ≤ KJk_diff + KJc_diff * rank t
    if t ∈ set (split_parts t2 t1) for t
  proof −
    have inv_t: inv t using split_parts_inv[OF assms(1)] that by simp
   have (2 + K_min) * (c_u * log 2 (size1 t)) ≤ (2 + K_min) * ((c_u / c_l) * rank
t + c_u)
      using log_size1_le_rank_ratio[OF inv_t] K_min_nonneg by (auto intro:
mult_left_mono)
    then show ?thesis by argo
  qed
  have join_work_diff t1 t2 ≤ join_work_diff' t1 t2
    using join_work_diff_le_diff'[OF assms(1,2)] .
  also have ... ≤ (∑ t ← split_parts t2 t1. 2 + c_u + (2 + K_min) * c_u * log 2
(size1 t))
    using join_work_diff'_as_sum[OF assms(1,2)] .
  also have ... ≤ (∑ t ← split_parts t2 t1. KJk_diff + KJc_diff * rank t)
    using pw by (intro sum_list_mono) auto
  finally show ?thesis .
qed
```

9.2 Final bound

```
time_fun diff equations diff.simps
declare T_diff.simps[simp del]

lemma T_diff_bridge:
  ⟦inv t1; inv t2⟧ ⟹ real (T_diff t1 t2) ≤ K_T * W_diff t1 t2
proof (induction t1 t2 rule: diff.induct)
  case (1 t1 t2)
  then show ?case
    using K_T_ge_1
     by (fastforce simp: T_diff.simps[of t1 t2] W_diff.simps[of t1 t2] inv_diff
split_inv
                  simp del: T_join2.simps
                  intro!: rule_cost_step_inter2 T_split_bridge
                  split: tree.splits prod.splits)
```

qed

lemma *KJk_diff_pos*: *0 < KJk_diff*
using *KJk_inter_pos* **by** *fastforce*

lemma *KJc_diff_pos*: *0 < KJc_diff*
using *KJc_inter_pos* **by** *fastforce*

theorem *W_diff_bound*:
assumes *inv t1 inv t2*
shows *W_diff t1 t2*
≤ 1 + K_lin KJk_diff KJc_diff ∗ size_min t1 t2
+ K_log KJk_diff KJc_diff
∗ (size_min t1 t2 ∗ log 2 (size_max t1 t2 / size_min t1 t2 + 1))
using *W_bound_generic'[OF assms(2,1)*
W_diff_work_bound[OF assms]
join_work_diff_le_sum_rank[OF assms]
KJk_diff_pos KJc_diff_pos]
by (*simp add*: *min.commute max.commute*)

corollary *T_diff_bound*:
assumes *inv t1 inv t2*
shows *real (T_diff t1 t2)*
≤ K_T + (K_T ∗ K_lin KJk_diff KJc_diff) ∗ size_min t1 t2
+ (K_T ∗ K_log KJk_diff KJc_diff)
∗ (size_min t1 t2 ∗ log 2 (size_max t1 t2 / size_min t1 t2 + 1))
proof −
have *real (T_diff t1 t2) ≤ K_T ∗ W_diff t1 t2*
using *T_diff_bridge assms(1,2)* **by** *simp*
also have *... ≤ K_T ∗ (1 + K_lin KJk_diff KJc_diff ∗ size_min t1 t2 +*
K_log KJk_diff KJc_diff ∗ (size_min t1 t2 ∗ log 2 (size_max t1 t2 / size_min t1 t2 + 1)))
using *W_diff_bound[OF assms] K_T_ge_1* **by** *auto*
also have *... = K_T + (K_T ∗ K_lin KJk_diff KJc_diff) ∗ size_min t1 t2 +*
(K_T ∗ K_log KJk_diff KJc_diff)
∗ (size_min t1 t2 ∗ log 2 (size_max t1 t2 / size_min t1 t2 + 1))
by *argo*
finally show *?thesis* **.**
qed

lemma *W_diff_nonneg*: *0 ≤ W_diff t1 t2*
proof (*induction t1 t2 rule*: *W_diff.induct*)
case (*1 t1 t2*)
then show *?case*
by (*auto simp*: *W_diff.simps[of t1 t2]*
intro!: *add_nonneg_nonneg W_split_nonneg W_join2_nonneg*
split: *tree.splits prod.splits*)

qed

As both the product filter and the bounding function are symmetric, the result is transported along when swapping the terms.

lemma *filterlim_swap_tree_pair*:
filterlim prod.swap tree_pair_filter tree_pair_filter
proof −
have *eventually* (λ*x*. *P* (*prod.swap x*)) *tree_pair_filter*
if *eventually P tree_pair_filter*
for *P* :: (('*a* × '*b*) *tree* × ('*a* × '*b*) *tree*) ⇒ *bool*
proof −
from *that* **obtain** *Pf Pg* **where** *eventually Pf tree_filter eventually Pg tree_filter*
⋀*x y*. *Pf x* ⟹ *Pg y* ⟹ *P* (*x*, *y*)
unfolding *tree_pair_filter_def eventually_prod_filter* **by** *blast*
— The witnesses are handed back in swapped order
then have *eventually Pg tree_filter* ∧ *eventually Pf tree_filter*
∧ (∀ *x y*. *Pg x* ⟶ *Pf y* ⟶ *P* (*prod.swap* (*x*, *y*)))
by *simp*
then show *?thesis*
unfolding *tree_pair_filter_def eventually_prod_filter* **by** *blast*
qed
then show *?thesis*
unfolding *filterlim_iff* **by** *blast*
qed

theorem *W_diff_bigo*:
(λ(*t1*, *t2*). *W_diff t1 t2*) ∈ *O*[*tree_pair_filter*](λ(*t1*,*t2*).
size_min t1 t2 ∗ *ln* (*size_max t1 t2* / *size_min t1 t2* + *1*))
proof −
have (λ(*ta*, *tb*). *W_diff tb ta*) ∈ *O*[*tree_pair_filter*](λ(*ta*, *tb*).
size_min ta tb ∗ *ln* (*size_max ta tb* / *size_min ta tb* + *1*))
proof −
have ⋀*ta tb*. *0* ≤ *W_diff tb ta*
by (*rule W_diff_nonneg*)
moreover have ⋀*ta tb*. ⟦*inv ta*; *inv tb*⟧ ⟹ *W_diff tb ta*
≤ *1* + *length* (*split_parts ta tb*) + *join_work_diff tb ta* + *split_work ta tb*
using *W_diff_work_bound* **by** *blast*
moreover have ⋀*ta tb*. ⟦*inv ta*; *inv tb*⟧
⟹ *join_work_diff tb ta* ≤ (∑ *t* ← *split_parts ta tb*. *KJk_diff* + *KJc_diff*
∗ *rank t*)
using *join_work_diff_le_sum_rank* **by** *blast*
ultimately show *?thesis*
using *KJk_diff_pos KJc_diff_pos*
W_bound_bigo_generic_ln[**where** *k*=*KJk_diff* **and** *c*=*KJc_diff*] **by**
force
qed
then have (λ*x*. (λ(*ta*, *tb*). *W_diff tb ta*) (*prod.swap x*))
∈ *O*[*tree_pair_filter*](λ*x*. (λ(*ta*, *tb*).
size_min ta tb ∗ *ln* (*size_max ta tb* / *size_min ta tb* + *1*)) (*prod.swap x*))

using *landau_o.big.compose filterlim_swap_tree_pair* **by** *blast*
then show *?thesis*
by (*simp add*: *min.commute max.commute*)
qed

corollary *T_diff_bigo*:
(λ(*t1*, *t2*). *real* (*T_diff t1 t2*)) ∈ *O*[*tree_pair_filter*](λ(*t1*,*t2*).
size_min t1 t2 ∗ *ln* (*size_max t1 t2* / *size_min t1 t2* + *1*))
proof −
have *eventually* (λ*x*. *norm* ((λ(*t1*, *t2*). *real* (*T_diff t1 t2*)) *x*)
≤ *K_T* ∗ *norm* ((λ(*t1*, *t2*). *W_diff t1 t2*) *x*)) *tree_pair_filter*
unfolding *tree_pair_filter_def* **using** *eventually_tree_pair_filter*
by (*auto simp*: *W_diff_nonneg elim*!: *eventually_mono intro*!: *T_diff_bridge*)
then have (λ(*t1*, *t2*). *real* (*T_diff t1 t2*))
∈ *O*[*tree_pair_filter*](λ(*t1*, *t2*). *W_diff t1 t2*)
using *bigoI* **by** *fast*
then show *?thesis*
using *W_diff_bigo landau_o.big_trans* **by** *blast*
qed

end

end

10 Strongly Joinable AVL Trees

theory *StronglyJoinableAVL*
imports
StronglyJoinable
HOL−Data_Structures.AVL_Set
begin

As a proof of concept, the *StronglyJoinable* framework is instantiated for the AVL trees of the standard library, with the height as rank and balance constants $c_l = 1$ and $c_u = 2$. Instantiating the abstract results then yields fully numeric running-time bounds for union, intersection and difference on AVL trees.

10.1 Join for AVL Trees

Following [4], *join* walks down the spine of the taller tree until it reaches a subtree whose height is within one of the smaller tree, attaches the smaller tree together with the pivot there, and rebalances on the way back up with the standard AVL rotations (*balL*, *balR*).

fun *joinR* **where**
joinR L a R = (*case L of Node l* (*k*,_) *r* ⇒
if ht r ≤ *ht R* + *1*

then balR l k (node r a R)
else balR l k (joinR r a R))

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

fun *joinL* **where**
joinL L a R = (case R of Node l (k,_) r ⇒
if ht l ≤ ht L + 1
then balL (node L a l) k r
else balL (joinL L a l) k r)

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

fun *join* **where**
join l a r =
(if ht l > ht r + 1 then joinR l a r
else if ht r > ht l + 1 then joinL l a r
else node l a r)

10.2 Proof of Correctness

The *Set2_Join* interpretation requires the set semantics of *join*, preservation of the search-tree order, and preservation of the AVL invariant.

10.2.1 Set Preservation

lemma *balR_set*:*set_tree (balR l a r) = set_tree*
l ∪ {a} ∪ set_tree r
by (*auto simp*: *balR_def node_def split*!: *if_splits tree.splits*)

lemma *joinR_set*:*ht l > ht r + 1 ⟹ set_tree (joinR l a r) = set_tree*
l ∪ {a} ∪ set_tree r
proof(*induction l a r rule*: *joinR.induct*)
case (*1 L a R*)
then show *?case*
by (*auto simp*: *joinR.simps*[*of L a R*] *balR_set node_def split*!: *if_splits tree.splits*)
qed

lemma *balL_set*:*set_tree (balL l a r) = set_tree*
l ∪ {a} ∪ set_tree r
by (*auto simp*: *balL_def node_def split*!: *if_splits tree.splits*)

lemma *joinL_set*:*ht r > ht l + 1 ⟹ set_tree (joinL l a r) = set_tree*
l ∪ {a} ∪ set_tree r
proof(*induction l a r rule*: *joinL.induct*)
case (*1 L a R*)
then show *?case*

by (*auto simp*: *joinL.simps*[*of L a R*] *balL_set node_def split*!: *if_splits tree.splits*)
qed

corollary *set_join*:*set_tree* (*join l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
by(*auto simp*: *node_def joinR_set joinL_set split*!: *if_splits*)

10.2.2 BST Preservation

lemma *balR_bst*:⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *a*; ∀ *y*∈*set_tree r*. *a* < *y*⟧
⟹ *bst* (*balR l a r*)
by (*auto simp*: *balR_def node_def split*!: *if_splits tree.splits*)

lemma *joinR_bst*:⟦*ht l* > *ht r* + *1*; *bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *a*; ∀ *y*∈*set_tree r*. *a* < *y*⟧
⟹ *bst* (*joinR l a r*)
proof(*induction l a r arbitrary*: *b rule*: *joinR.induct*)
case (*1 L a R*)
then show *?case*
by (*auto simp*: *joinR.simps*[*of L a R*] *node_def balR_bst joinR_set ball_Un intro*!: *balR_bst split*!: *if_splits tree.splits*)
qed

lemma *balL_bst*:⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *a*; ∀ *y*∈*set_tree r*. *a* < *y*⟧
⟹ *bst* (*balL l a r*)
by (*auto simp*: *balL_def node_def split*!: *if_splits tree.splits*)

lemma *joinL_bst*:⟦*ht r* > *ht l* + *1*; *bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *a*; ∀ *y*∈*set_tree r*. *a* < *y*⟧
⟹ *bst* (*joinL l a r*)
proof(*induction l a r arbitrary*: *b rule*: *joinL.induct*)
case (*1 L a R*)
then show *?case*
by (*auto simp*: *joinL.simps*[*of L a R*] *node_def balL_bst joinL_set ball_Un intro*!: *balL_bst split*!: *if_splits tree.splits*)
qed

corollary *bst_join*:*bst* (*Node l* (*a*, *b*) *r*) ⟹ *bst* (*join l a r*)
by(*auto simp*: *node_def joinR_bst joinL_bst split*!: *if_splits*)

10.2.3 Preservation of AVL Predicate

lemma *avl_joinR_height*:⟦*avl l*; *avl r*; *height l* > *height r* + *1*⟧
⟹ *avl* (*joinR l a r*) ∧ *height* (*joinR l a r*) ∈ {*height l*, *height l* + *1*}
proof(*induction l a r rule*: *joinR.induct*)
case (*1 L a R*)
then show *?case*
by(*fastforce simp*: *joinR.simps*[*of L a R*] *balR_def node_def max_absorb2 split*!: *if_splits tree.splits*)
qed

```
lemma avl_joinL_height:⟦avl l; avl r; height r > height l + 1⟧
⟹ avl (joinL l a r) ∧ height (joinL l a r) ∈ {height r, height r + 1}
proof(induction l a r rule: joinL.induct)
  case (1 L a R)
  then show ?case
    by(fastforce simp: joinL.simps[of L a R] balL_def node_def max_absorb2 split!:
if_splits tree.splits)
qed

corollary inv_join:⟦avl l; avl r⟧ ⟹ avl (join l a r)
  by(auto simp: node_def avl_joinR_height avl_joinL_height split: if_splits)
```

To finish the proof of functional correctness, instantiate the *Set2_Join* locale.

```
interpretation tree_ht: Set2_Join
where join = join and inv = avl
proof (standard, goal_cases)
  case 1 show ?case by (rule set_join)
next
  case 2 then show ?case by (rule bst_join)
next
  case 3 show ?case by simp
next
  case 4 then show ?case by (rule inv_join)
next
  case 5 then show ?case by simp
qed
```

10.3 Proof of Strongly Joinable Properties

The additional obligations of *StronglyJoinable* are discharged with *height* as rank. The central fact is *join_height* below i.e. a join has the height of its taller argument or one more. Monotonicity and the submodularity rule are elementary consequences. The balance predicate holds for AVL trees with $c_l = 1$ and $c_u = 2$.

10.3.1 Monotonicity Rule

```
corollary join_height:
assumes avl l ∧ avl r
shows height (join l a r) ∈ {max (height l) (height r), max (height l) (height r)
+ 1}
proof−
  consider
    (Right) R  where R = joinR l a r and ht l > ht r + 1 and
                    join l a r = R
  | (Left)  L  where L = joinL l a r and ht r > ht l + 1 and
```

```
                join l a r = L
  | (Node)  EQ where EQ = node l a r and ht r ≤ ht l + 1 and
                ht l ≤ ht r + 1 and join l a r = EQ
    by(auto split!: if_splits)linarith
  then show ?thesis
  proof cases
    case Right
    from this assms show ?thesis
      by (metis add_lessD1 avl_joinR_height ht_height max.absorb3)
  next
    case Left
    from this assms show ?thesis
      by (metis add_lessD1 ht_height avl_joinL_height max.absorb4)
  next
    case Node
    then show ?thesis
      by simp
  qed
qed

corollary rule_mono: ⟦avl l; avl r⟧ ⟹ max (height l) (height r) ≤ height (join l
a r)
  using join_height
  by (metis insertE le_add_same_cancel1 zero_le_one nle_le singletonD)
```

10.3.2 Submodularity Rule

A join adds at most one to the larger input height, while a node adds exactly one to the larger child height. Hence, whatever the replacement subtrees are, the join can exceed the original node by no more than the replacements exceed the original subtrees. This also covers the mixed case where one subtree shrinks.

```
lemma rule_sub:
  assumes real (height l') ≤ real (height l) + x real (height r') ≤ real (height r)
+ x
  assumes avl l' avl r'
  shows real (height (join l' a r')) ≤ real (height (Node l (a,b) r)) + x
proof −
  have height (join l' a r') ≤ max (height l') (height r') + 1
    using join_height[of l' r' a] assms(3,4) by auto
  then have real (height (join l' a r')) ≤ max (real (height l')) (real (height r'))
+ 1
    by (metis of_nat_le_iff of_nat_max of_nat_1 of_nat_add)
  moreover have max (real (height l')) (real (height r')) ≤ max (real (height l))
(real (height r)) + x
    using assms(1,2) by linarith
  moreover have real (height (Node l (a,b) r)) = max (real (height l)) (real (height
r)) + 1
```

by *force*
ultimately show *?thesis*
by *linarith*
qed

10.3.3 Balancing Rule

interpretation *tree_ht*: *BalancedShape*
where *rank* = *height* **and** c_l = *1* **and** c_u = *2*
by (*standard*, *goal_cases*) *auto*

lemma *rule_bal*: *avl t* $\Longrightarrow$ *tree_ht.balanced t*
by(*induction t*) *auto*

interpretation *tree_ht*: *BalancedTree*
where *rank* = *height* **and** c_l = *1* **and** c_u = *2* **and** *inv* = *avl*
by *standard* (*rule rule_bal*)

10.3.4 Cost Rule

time_fun *ht*
time_fun *node*
time_fun *balL*
time_fun *balR*

lemma *T_ht_0*[*simp*]: *T_ht t* = *0*
by (*cases t rule*: *tree2_cases*) *auto*

lemma *T_node_0*[*simp*]: *T_node l a r* = *0*
by *simp*

lemma *T_balL_0*[*simp*]: *T_balL l a r* = *0*
by (*auto split*!: *tree.splits if_splits*)

lemma *T_balR_0*[*simp*]: *T_balR l a r* = *0*
by (*auto split*!: *tree.splits if_splits*)

time_fun *joinR*
time_fun *joinL*
time_fun *join*

declare *T_joinR.simps*[*simp del*]
declare *T_joinL.simps*[*simp del*]

lemma *T_joinR*:⟦*avl l*; *avl r*; *ht l* > *ht r* + *1*⟧ $\Longrightarrow$ *T_joinR l a r* $\leq$ *1* + *height l* − *height r*
proof(*induction l a r rule*: *T_joinR.induct*)
case (*1 L a R*)
then show *?case*
using *T_joinR.simps*[*of L a R*]

```
  by(auto simp: max_absorb2 split!: if_splits tree.splits) fastforce+
qed

lemma T_joinL:⟦avl l; avl r; ht r > ht l + 1⟧ ⟹ T_joinL l a r ≤ 1 + height r
− height l
proof(induction l a r rule: T_joinL.induct)
 case (1 L a R)
 then show ?case
  by(auto simp: T_joinL.simps[of L a R] max_absorb1 split!: if_splits tree.splits)
fastforce+
qed

corollary rule_cost:
assumes avl l ∧ avl r
shows T_join l a r ≤ 1 + nat(abs(int(height l) − int(height r)))
proof−
 consider
   (Right) R where R = T_joinR l a r and ht l > ht r + 1 and
                T_join l a r = R
 | (Left)  L where L = T_joinL l a r and ht r > ht l + 1 and
                T_join l a r = L
 | (Node)  EQ where EQ = 0 and ht r ≤ ht l + 1 and
                 ht l ≤ ht r + 1 and T_join l a r = EQ
   by(auto split!: if_splits)linarith
 then show ?thesis
 proof cases
   case Right
   then have max (height l) (height r) = height l
     using assms by simp
    then moreover have nat(abs(int(height l) − int(height r))) = height l −
height r
     by simp
   ultimately show ?thesis
     using Right T_joinR assms by (metis Nat.add_diff_assoc max.orderI)
 next
   case Left
   then have max (height l) (height r) = height r
     using assms by simp
    then moreover have nat(abs(int(height l) − int(height r))) = height r −
height l
     by simp
   ultimately show ?thesis
     using Left T_joinL assms by (metis Nat.add_diff_assoc max.cobounded1)
 next
   case Node
   then show ?thesis
     by simp
 qed
qed
```

10.4 Instantiation of *StronglyJoinable*

By functional correctness, every join adds exactly one element.

lemma *balR_size*:*size* (*balR l a r*) = *size l* + *size r* + *1*
  by(*auto simp*: *balR_def node_def split*!: *if_split tree.split*)

lemma *balL_size*:*size* (*balL l a r*) = *size l* + *size r* + *1*
  by(*auto simp*: *balL_def node_def split*!: *if_split tree.split*)

lemma *joinR_size*:*ht l* > *ht r* + *1* ⟹ *size* (*joinR l a r*) = *size l* + *size r* + *1*
proof(*induction l a r rule*: *joinR.induct*)
  case (*1 L a R*)
  then show *?case*
    by (*auto simp*: *joinR.simps*[*of L a R*] *balR_size node_def split*!: *if_splits tree.splits*)
qed

lemma *joinL_size*:*ht r* > *ht l* + *1* ⟹ *size* (*joinL l a r*) = *size l* + *size r* + *1*
proof(*induction l a r rule*: *joinL.induct*)
  case (*1 L a R*)
  then show *?case*
    by (*auto simp*: *joinL.simps*[*of L a R*] *balL_size node_def split*!: *if_splits tree.splits*)
qed

lemma *join_size*:*size* (*join l a r*) = *size l* + *size r* + *1*
  by(*auto simp*: *node_def joinR_size joinL_size split*!: *if_splits*)

All obligations of *StronglyJoinable* are now in place.

interpretation *tree_ht*: *StronglyJoinable*
where *join* = *join* **and** *inv* = *avl*
and *rank* = *height* **and** c_l = *1* **and** c_u = *2*
and *T_join* = *T_join* **and** k_0 = *1* **and** k_1 = *1*
proof (*standard*, *goal_cases*)
  case (*1 l r a*)
  then show *?case*
    by (*metis of_nat_le_iff of_nat_max rule_mono*)
next
  case (*2 l' l x r' r a b*)
  then show *?case*
    by (*intro rule_sub*)
next
  case (*3 l a r*)
  then show *?case*
    by (*metis join_size*)
next
  case (*4 l r a*)
  have *real* (*T_join l a r*) ≤ *1* + *real* (*nat* |*int* (*height l*) − *int* (*height r*)|)
    using *rule_cost 4* **by** (*metis of_nat_1 of_nat_add of_nat_le_iff*)

```
  also have real (nat |int (height l) − int (height r)|)
          = |real (height l) − real (height r)|
    by linarith
  finally show ?case by simp
next
  case 5
  then show ?case by simp
qed
```

10.5 Concrete constants for AVL

All abstract constants are evaluated for the AVL instantiation.

```
lemma avl_c_delta: tree_ht.c_δ = 1
  unfolding tree_ht.c_δ_def by simp

lemma avl_C: tree_ht.C = 1
  unfolding tree_ht.C_def by simp

lemma avl_K_split: tree_ht.K_split = 8
  unfolding tree_ht.K_split_def by simp

lemma avl_K_T: tree_ht.K_T = 4
  unfolding tree_ht.K_T_def by simp

lemma avl_K_min: tree_ht.K_min = 4
  unfolding tree_ht.c₁_def avl_c_delta tree_ht.K_min_def by auto

lemma avl_K_lin_union: tree_ht.K_T * tree_ht.K_lin 2 3 ≤ 1238
proof −
  have tree_ht.K_T * tree_ht.K_lin 2 3 = 756 + 340 * sqrt 2
    unfolding tree_ht.K_lin_def
    by (simp add: avl_K_T avl_C avl_K_split S1_two S2_two algebra_simps)
  also have ... ≤ 756 + 340 * 1.415
    using sqrt2_bounds(3) by (auto intro: add_left_mono mult_left_mono)
  also have 756 + 340 * (1.415::real) ≤ 1238
    by simp
  finally show ?thesis .
qed

lemma avl_K_log_union: tree_ht.K_T * tree_ht.K_log 2 3 ≤ 629
proof −
  have tree_ht.K_T * tree_ht.K_log 2 3 = 504 + 88 * sqrt 2
    unfolding tree_ht.K_log_def
    by (simp add: avl_K_T avl_C avl_K_split S1_two algebra_simps)
  also have ... ≤ 504 + 88 * 1.415
    using sqrt2_bounds(3) by (auto intro: add_left_mono mult_left_mono)
  also have 504 + 88 * (1.415::real) ≤ 629
    by simp
  finally show ?thesis .
```

qed

lemma *avl_K_lin_inter*: *tree_ht.K_T * tree_ht.K_lin 16 12 ≤ 2222*
proof −
  have *tree_ht.K_T * tree_ht.K_lin 16 12 = 1356 + 612 * sqrt 2*
    unfolding *tree_ht.K_lin_def*
    by (*simp add*: *avl_K_T avl_C avl_K_split S1_two S2_two algebra_simps*)
  also have *... ≤ 1356 + 612 * 1.415*
    using *sqrt2_bounds*(*3*) **by** (*auto intro*: *add_left_mono mult_left_mono*)
  also have *1356 + 612 * (1.415*::*real*) *≤ 2222*
    by *simp*
  finally show *?thesis* **.**
qed

lemma *avl_K_log_inter*: *tree_ht.K_T * tree_ht.K_log 16 12 ≤ 1131*
proof −
  have *tree_ht.K_T * tree_ht.K_log 16 12 = 904 + 160 * sqrt 2*
    unfolding *tree_ht.K_log_def*
    by (*simp add*: *avl_K_T avl_C avl_K_split S1_two algebra_simps*)
  also have *... ≤ 904 + 160 * 1.415*
    using *sqrt2_bounds*(*3*) **by** (*auto intro*: *add_left_mono mult_left_mono*)
  also have *904 + 160 * (1.415*::*real*) *≤ 1131*
    by *simp*
  finally show *?thesis* **.**
qed

This yields fully numeric running-time bounds for the three set operations on AVL trees.

corollary *avl_T_union_linlog*:
  assumes *avl t1 avl t2*
  shows *real* (*tree_ht.T_union t1 t2*)
      *≤ 4 + 1238 * size_min t1 t2*
        *+ 629 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))*
proof −
  have *real* (*tree_ht.T_union t1 t2*)
      *≤ tree_ht.K_T + (tree_ht.K_T * tree_ht.K_lin 2 3) * size_min t1 t2*
        *+ (tree_ht.K_T * tree_ht.K_log 2 3)*
          ** (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))*
    using *tree_ht.T_union_bound*[*OF assms*]
    by (*simp add*: *avl_c_delta*)
  also have *... ≤ 4 + 1238 * size_min t1 t2*
      *+ 629 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))*
    using *avl_K_lin_union avl_K_log_union avl_K_T*
    by (*intro add_mono mult_right_mono*) (*auto simp*: *log_def*)
  finally show *?thesis* **.**
qed

corollary *avl_T_inter_linlog*:
  assumes *avl t1 avl t2*

```
  shows real (tree_ht.T_inter t1 t2)
      ≤ 4 + 2222 * size_min t1 t2
        + 1131 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have real (tree_ht.T_inter t1 t2)
     ≤ tree_ht.K_T + (tree_ht.K_T * tree_ht.K_lin 16 12) * size_min t1 t2
       + (tree_ht.K_T * tree_ht.K_log 16 12)
         * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using tree_ht.T_inter_bound[OF assms]
    by (simp add: avl_K_min)
  also have ... ≤ 4 + 2222 * size_min t1 t2
       + 1131 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using avl_K_lin_inter avl_K_log_inter avl_K_T
    by (intro add_mono mult_right_mono) (auto simp: log_def)
  finally show ?thesis .
qed

corollary avl_T_diff_linlog:
  assumes avl t1 avl t2
  shows real (tree_ht.T_diff t1 t2)
      ≤ 4 + 2222 * size_min t1 t2
        + 1131 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have real (tree_ht.T_diff t1 t2)
     ≤ tree_ht.K_T + (tree_ht.K_T * tree_ht.K_lin 16 12) * size_min t1 t2
       + (tree_ht.K_T * tree_ht.K_log 16 12)
         * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using tree_ht.T_diff_bound[OF assms]
    by (simp add: avl_K_min)
  also have ... ≤ 4 + 2222 * size_min t1 t2
       + 1131 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using avl_K_lin_inter avl_K_log_inter avl_K_T
    by (intro add_mono mult_right_mono) (auto simp: log_def)
  finally show ?thesis .
qed
```

The asymptotic statements are inherited through the interpretation. On pairs of growing AVL trees of sizes $m \le n$, in either argument order, all three operations run in $O(m \ln(n/m + 1))$.

```
corollary avl_T_union_bigo:
  (λ(t1, t2). real (tree_ht.T_union t1 t2)) ∈ O[tree_ht.tree_pair_filter](λ(t1, t2).
     size_min t1 t2 * ln (size_max t1 t2 / size_min t1 t2 + 1))
  using tree_ht.T_union_bigo .

corollary avl_T_inter_bigo:
  (λ(t1, t2). real (tree_ht.T_inter t1 t2)) ∈ O[tree_ht.tree_pair_filter](λ(t1, t2).
     size_min t1 t2 * ln (size_max t1 t2 / size_min t1 t2 + 1))
  using tree_ht.T_inter_bigo .
```

corollary *avl_T_diff_bigo*:
(λ(*t1*, *t2*). *real* (*tree_ht.T_diff t1 t2*)) ∈ *O*[*tree_ht.tree_pair_filter*](λ(*t1*, *t2*).
size_min t1 t2 ∗ *ln* (*size_max t1 t2* / *size_min t1 t2* + *1*))
using *tree_ht.T_diff_bigo* **.**

end

11 Strongly Joinable Red-Black Trees

theory *StronglyJoinableRBT*
imports
StronglyJoinable
begin

The second instantiation of *StronglyJoinable* covers red-black trees. The *join* of [4] stores the black height in every node and reads it in constant time, so the metadata field of the trees holds the colour together with the black height. The library implementation in *HOL−Data_Structures.Set2_Join_RBT* recomputes the black height on every step instead, which would charge *join* a cost linear in the ranks of its inputs rather than in their difference. Its rebalancing steps are, however, taken over unchanged.

Two further deviations from the library concern the colour of the root. The library *join* paints the root black unconditionally and creates a black node whenever the two inputs have equal black height. Following the paper, the root is painted black only when it is red with a red child, and two black inputs of equal black height are combined by a red node. Both are necessary for the rank of a join to exceed the larger input rank by at most one.

11.1 Red-black trees with stored black height

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

type_synonym *'a rbt* = (*'a* ∗ (*color* ∗ *nat*)) *tree*

Both the colour and the black height are read from the node.

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

fun *bh* :: *'a rbt* ⇒ *nat* **where**
bh Leaf = *0* |
bh (*Node* _ (_, (_, *h*)) _) = *h*

fun *R* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
R l a r = *Node l* (*a*, (*Red*, *bh l*)) *r*

fun $B :: 'a\ rbt \Rightarrow 'a \Rightarrow 'a\ rbt \Rightarrow 'a\ rbt$ **where**
$B\ l\ a\ r = Node\ l\ (a, (Black, bh\ l + 1))\ r$

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

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

11.1.1 Invariants

fun $invc :: 'a\ rbt \Rightarrow bool$ **where**
$invc\ Leaf = True\ |$
$invc\ (Node\ l\ (\_, (c, \_))\ r) = ((c = Red \longrightarrow col\ l = Black \wedge col\ r = Black) \wedge invc\ l \wedge invc\ r)$

The weaker colour invariant allows a red root with a red child.

fun $invc2 :: 'a\ rbt \Rightarrow bool$ **where**
$invc2\ Leaf = True\ |$
$invc2\ (Node\ l\ \_\ r) = (invc\ l \wedge invc\ r)$

fun $invh :: 'a\ rbt \Rightarrow bool$ **where**
$invh\ Leaf = True\ |$
$invh\ (Node\ l\ (\_, (c, h))\ r) =$
$(bh\ l = bh\ r \wedge h = (if\ c = Black\ then\ bh\ l + 1\ else\ bh\ l) \wedge invh\ l \wedge invh\ r)$

abbreviation $rbt :: 'a\ rbt \Rightarrow bool$ **where**
$rbt\ t \equiv invc\ t \wedge invh\ t$

lemma *invc2I*: $invc\ t \Longrightarrow invc2\ t$
by (*cases t rule*: *tree2_cases*) *auto*

11.1.2 Rebalancing

The rotations *baliL* and *baliR* of the library, restated for the stored black heights.

fun $baliL :: 'a\ rbt \Rightarrow 'a \Rightarrow 'a\ rbt \Rightarrow 'a\ rbt$ **where**
$baliL\ (Node\ (Node\ t1\ (a, (Red, \_))\ t2)\ (b, (Red, \_))\ t3)\ c\ t4 = R\ (B\ t1\ a\ t2)\ b\ (B\ t3\ c\ t4)\ |$
$baliL\ (Node\ t1\ (a, (Red, \_))\ (Node\ t2\ (b, (Red, \_))\ t3))\ c\ t4 = R\ (B\ t1\ a\ t2)\ b\ (B\ t3\ c\ t4)\ |$
$baliL\ t1\ a\ t2 = B\ t1\ a\ t2$

fun $baliR :: 'a\ rbt \Rightarrow 'a \Rightarrow 'a\ rbt \Rightarrow 'a\ rbt$ **where**
$baliR\ t1\ a\ (Node\ t2\ (b, (Red, \_))\ (Node\ t3\ (c, (Red, \_))\ t4)) = R\ (B\ t1\ a\ t2)\ b\ (B\ t3\ c\ t4)\ |$
$baliR\ t1\ a\ (Node\ (Node\ t2\ (b, (Red, \_))\ t3)\ (c, (Red, \_))\ t4) = R\ (B\ t1\ a\ t2)\ b\ (B\ t3\ c\ t4)\ |$
$baliR\ t1\ a\ t2 = B\ t1\ a\ t2$

lemma *inv_baliL*:
⟦ *invh l*; *invh r*; *invc2 l*; *invc r*; *bh l* = *bh r* ⟧
⟹ *invc* (*baliL l a r*) ∧ *invh* (*baliL l a r*) ∧ *bh* (*baliL l a r*) = *bh l* + *1*
by (*induct l a r rule*: *baliL.induct*) *auto*

lemma *inv_baliR*:
⟦ *invh l*; *invh r*; *invc l*; *invc2 r*; *bh l* = *bh r* ⟧
⟹ *invc* (*baliR l a r*) ∧ *invh* (*baliR l a r*) ∧ *bh* (*baliR l a r*) = *bh l* + *1*
by (*induct l a r rule*: *baliR.induct*) *auto*

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_baliR*: *set_tree* (*baliR l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
by (*cases* (*l*,*a*,*r*) *rule*: *baliR.cases*) *auto*

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

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

lemma *size_baliL*: *size* (*baliL l a r*) = *size l* + *size r* + *1*
by (*cases* (*l*,*a*,*r*) *rule*: *baliL.cases*) *auto*

lemma *size_baliR*: *size* (*baliR l a r*) = *size l* + *size r* + *1*
by (*cases* (*l*,*a*,*r*) *rule*: *baliR.cases*) *auto*

11.2 Join for Red-Black Trees

fun *joinL* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
joinL l x r =
(*if bh r* ≤ *bh l then R l x r*
else case r of Node l' (*x'*, (*c*, _)) *r'* ⇒
(*if c* = *Black then baliL* (*joinL l x l'*) *x' r' else R* (*joinL l x l'*) *x' r'*))

fun *joinR* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
joinR l x r =
(*if bh l* ≤ *bh r then R l x r*
else case l of Node l' (*x'*, (*c*, _)) *r'* ⇒
(*if c* = *Black then baliR l' x'* (*joinR r' x r*) *else R l' x'* (*joinR r' x r*)))

declare *joinL.simps*[*simp del*]
declare *joinR.simps*[*simp del*]

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

```
blacken (Node l (a, (Red, h)) r) =
  (if col l = Red ∨ col r = Red then B l a r else Node l (a, (Red, h)) r) |
blacken t = t

fun join :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where
join l x r =
  (if bh r < bh l then blacken (joinR l x r)
   else if bh l < bh r then blacken (joinL l x r)
   else if col l = Black ∧ col r = Black then R l x r else B l x r)
```

11.3 Proof of Correctness

11.3.1 Colour and height invariants

```
lemma inv_joinL:
  ⟦ invc l; invc r; invh l; invh r; bh l ≤ bh r ⟧ ⟹
   invc2 (joinL l x r) ∧ (bh l ≠ bh r ∧ col r = Black ⟶ invc (joinL l x r))
   ∧ invh (joinL l x r) ∧ bh (joinL l x r) = bh r
proof (induct l x r rule: joinL.induct)
  case (1 l x r)
  then show ?case
    by (auto simp: inv_baliL invc2I joinL.simps[of l x r] split!: tree.splits if_splits)
qed

lemma inv_joinR:
  ⟦ invc l; invc r; invh l; invh r; bh r ≤ bh l ⟧ ⟹
   invc2 (joinR l x r) ∧ (bh l ≠ bh r ∧ col l = Black ⟶ invc (joinR l x r))
   ∧ invh (joinR l x r) ∧ bh (joinR l x r) = bh l
proof (induct l x r rule: joinR.induct)
  case (1 l x r)
  then show ?case
    by (auto simp: inv_baliR invc2I joinR.simps[of l x r] split!: tree.splits if_splits)
qed
```

A red root is passed upwards unchanged.

```
lemma col_joinL: ⟦bh l < bh r; col r = Red⟧ ⟹ col (joinL l x r) = Red
  by (auto simp: joinL.simps[of l x r] split!: tree.splits if_splits)

lemma col_joinR: ⟦bh r < bh l; col l = Red⟧ ⟹ col (joinR l x r) = Red
  by (auto simp: joinR.simps[of l x r] split!: tree.splits if_splits)

lemma inv_blacken: ⟦invc2 t; invh t⟧ ⟹ invc (blacken t) ∧ invh (blacken t)
  by (cases t rule: blacken.cases) auto

lemma blacken_id: invc t ⟹ blacken t = t
  by (cases t rule: blacken.cases) auto

lemma inv_join:
  assumes rbt l rbt r
  shows rbt (join l x r)
```

proof −
  have *rbt* (*blacken* (*joinR l x r*)) **if** *bh r* < *bh l*
    using *inv_joinR*[*of l r x*] *inv_blacken*[*of joinR l x r*] *assms that* **by** *auto*
  moreover have *rbt* (*blacken* (*joinL l x r*)) **if** *bh l* < *bh r*
    using *inv_joinL*[*of l r x*] *inv_blacken*[*of joinL l x r*] *assms that* **by** *auto*
  ultimately show *?thesis*
    using *assms* **by** *auto*
qed

11.3.2 Set and search-tree properties

lemma *set_joinL*: *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*)
  then show *?case*
    by (*auto simp*: *set_baliL joinL.simps*[*of l x r*] *split*!: *tree.splits if_splits*)
qed

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*)
  then show *?case*
    by (*auto simp*: *set_baliR joinR.simps*[*of l x r*] *split*!: *tree.splits if_splits*)
qed

lemma *set_blacken*: *set_tree* (*blacken t*) = *set_tree t*
  by (*cases t rule*: *blacken.cases*) *auto*

lemma *set_join*: *set_tree* (*join l x r*) = *set_tree l* ∪ {*x*} ∪ *set_tree r*
  by (*auto simp*: *set_joinL set_joinR set_blacken*)

lemma *bst_joinL*: *bst* (*Node l* (*a*, *n*) *r*) ⟹ *bst* (*joinL l a r*)
proof (*induction l a r rule*: *joinL.induct*)
  case (*1 l a r*)
  then show *?case*
    by (*auto simp*: *set_baliL joinL.simps*[*of l a r*] *set_joinL ball_Un intro*!: *bst_baliL*
        split!: *tree.splits if_splits*)
qed

lemma *bst_joinR*: *bst* (*Node l* (*a*, *n*) *r*) ⟹ *bst* (*joinR l a r*)
proof (*induction l a r rule*: *joinR.induct*)
  case (*1 l a r*)
  then show *?case*
    by (*auto simp*: *set_baliR joinR.simps*[*of l a r*] *set_joinR ball_Un intro*!: *bst_baliR*
        split!: *tree.splits if_splits*)
qed

lemma *bst_blacken*: *bst* (*blacken t*) = *bst t*
by (*cases t rule*: *blacken.cases*) *auto*

lemma *bst_join*: *bst* (*Node l* (*a*, *n*) *r*) ⟹ *bst* (*join l a r*)
by (*auto simp*: *bst_blacken bst_joinL bst_joinR*)

11.3.3 Size

lemma *size_joinL*: *size* (*joinL l x r*) = *size l* + *size r* + *1*
proof (*induction l x r rule*: *joinL.induct*)
case (*1 l x r*)
then show *?case*
by (*auto simp*: *size_baliL joinL.simps*[*of l x r*] *split*!: *tree.splits if_splits*)
qed

lemma *size_joinR*: *size* (*joinR l x r*) = *size l* + *size r* + *1*
proof (*induction l x r rule*: *joinR.induct*)
case (*1 l x r*)
then show *?case*
by (*auto simp*: *size_baliR joinR.simps*[*of l x r*] *split*!: *tree.splits if_splits*)
qed

lemma *size_blacken*: *size* (*blacken t*) = *size t*
by (*cases t rule*: *blacken.cases*) *auto*

lemma *join_size*: *size* (*join l x r*) = *size l* + *size r* + *1*
by (*auto simp*: *size_joinL size_joinR size_blacken*)

Functional correctness is established by instantiating *Set2_Join*.

interpretation *tree_rb*: *Set2_Join*
where *join* = *join* **and** *inv* = *rbt*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*rule set_join*)
next
case *2* **then show** *?case* **by** (*rule bst_join*)
next
case *3* **show** *?case* **by** *simp*
next
case *4* **then show** *?case* **by** (*rule inv_join*)
next
case (*5 l a b r*) **then show** *?case* **by** (*cases b*) *auto*
qed

11.4 Proof of Strongly Joinable Properties

definition *rk* :: *'a rbt* ⇒ *nat* **where**
rk t = *2* ∗ *bh t* + (*if col t* = *Red then 1 else 0*)

lemma *rk_Leaf*[*simp*]: *rk Leaf* = *0*
by (*simp add*: *rk_def*)

lemma *rk_Node*[*simp*]: *rk* (*Node l* (*a*, (*c*, *h*)) *r*) = *2* ∗ *h* + (*if c* = *Red then 1 else 0*)
by (*simp add*: *rk_def*)

11.4.1 Balancing Rule

lemma *rk_children*:
assumes *rbt* (*Node l* (*a*, (*c*, *h*)) *r*)
shows *max* (*rk l*) (*rk r*) + *1* ≤ *rk* (*Node l* (*a*, (*c*, *h*)) *r*)
and *rk* (*Node l* (*a*, (*c*, *h*)) *r*) ≤ *min* (*rk l*) (*rk r*) + *2*
using *assms* **by** (*cases c*; *cases col l*; *cases col r*; *simp add*: *rk_def*)+

interpretation *tree_rb*: *BalancedShape*
where *rank* = λ*t*. *real* (*rk t*) **and** c_l = *1* **and** c_u = *2*
by (*standard*, *goal_cases*) *auto*

lemma *rule_bal*: *rbt t* ⟹ *tree_rb.balanced t*
proof (*induction t rule*: *tree2_induct*)
case (*Node l a b r*)
obtain *c h* **where** *b*: *b* = (*c*, *h*)
by (*cases b*)
have *max* (*real* (*rk l*)) (*real* (*rk r*)) + *1* ≤ *real* (*rk* (*Node l* (*a*, *b*) *r*))
and *real* (*rk* (*Node l* (*a*, *b*) *r*)) ≤ *min* (*real* (*rk l*)) (*real* (*rk r*)) + *2*
using *rk_children*[*of l a c h r*] *Node.prems b*
by (*simp_all flip*: *of_nat_max of_nat_min*)
then show *?case*
using *Node b* **by** *auto*
qed *simp*

interpretation *tree_rb*: *BalancedTree*
where *rank* = λ*t*. *real* (*rk t*) **and** c_l = *1* **and** c_u = *2* **and** *inv* = *rbt*
by *standard* (*rule rule_bal*)

11.4.2 Monotonicity and Submodularity Rules

lemma *rk_blacken*:
assumes *invh t*
shows *rk t* ≤ *rk* (*blacken t*) **and** *rk* (*blacken t*) ≤ *2* ∗ *bh t* + *2*
using *assms* **by** (*cases t rule*: *blacken.cases*; *simp*)+

lemma *rk_joinR*:
assumes *rbt l rbt r bh r* < *bh l*
shows *max* (*rk l*) (*rk r*) ≤ *rk* (*blacken* (*joinR l x r*))
and *rk* (*blacken* (*joinR l x r*)) ≤ *max* (*rk l*) (*rk r*) + *1*
proof −
let *?t* = *joinR l x r*
have *t*: *invc2 ?t col l* = *Black* ⟶ *invc ?t invh ?t bh ?t* = *bh l*
using *inv_joinR*[*of l r x*] *assms* **by** *auto*
have *rk_t*: *rk ?t* = *2* ∗ *bh l* + (*if col ?t* = *Red then 1 else 0*)

```
    by (simp add: rk_def t(4))
  have rk_r: rk r ≤ 2 * bh l
    using assms(3) by (simp add: rk_def)
  show max (rk l) (rk r) ≤ rk (blacken ?t)
  proof (cases col l)
    case Red
    then have rk l = 2 * bh l + 1 and col ?t = Red
      using col_joinR[OF assms(3)] by (simp_all add: rk_def)
    then show ?thesis
      using rk_blacken(1)[OF t(3)] rk_t rk_r by simp
  next
    case Black
    then have rk l = 2 * bh l
      by (simp add: rk_def)
    then show ?thesis
      using rk_blacken(1)[OF t(3)] rk_t rk_r by simp
  qed
  show rk (blacken ?t) ≤ max (rk l) (rk r) + 1
  proof (cases col l)
    case Red
    then have rk l = 2 * bh l + 1
      by (simp add: rk_def)
    then show ?thesis
      using rk_blacken(2)[OF t(3)] t(4) max.cobounded1[of rk l rk r] by linarith
  next
    case Black
    then have rk l = 2 * bh l
      by (simp add: rk_def)
    moreover have rk (blacken ?t) = rk ?t
      using blacken_id[OF t(2)[THEN mp, OF Black]] by simp
    moreover have rk ?t ≤ 2 * bh l + 1
      using rk_t by simp
    ultimately show ?thesis
      using max.cobounded1[of rk l rk r] by linarith
  qed
qed

lemma rk_joinL:
  assumes rbt l rbt r bh l < bh r
  shows max (rk l) (rk r) ≤ rk (blacken (joinL l x r))
    and rk (blacken (joinL l x r)) ≤ max (rk l) (rk r) + 1
proof −
  let ?t = joinL l x r
  have t: invc2 ?t col r = Black ⟶ invc ?t invh ?t bh ?t = bh r
    using inv_joinL[of l r x] assms by auto
  have rk_t: rk ?t = 2 * bh r + (if col ?t = Red then 1 else 0)
    by (simp add: rk_def t(4))
  have rk_l: rk l ≤ 2 * bh r
    using assms(3) by (simp add: rk_def)
```

```
  show max (rk l) (rk r) ≤ rk (blacken ?t)
  proof (cases col r)
    case Red
    then have rk r = 2 * bh r + 1 and col ?t = Red
      using col_joinL[OF assms(3)] by (simp_all add: rk_def)
    then show ?thesis
      using rk_blacken(1)[OF t(3)] rk_t rk_l by simp
  next
    case Black
    then have rk r = 2 * bh r
      by (simp add: rk_def)
    then show ?thesis
      using rk_blacken(1)[OF t(3)] rk_t rk_l by simp
  qed
  show rk (blacken ?t) ≤ max (rk l) (rk r) + 1
  proof (cases col r)
    case Red
    then have rk r = 2 * bh r + 1
      by (simp add: rk_def)
    then show ?thesis
      using rk_blacken(2)[OF t(3)] t(4) max.cobounded2[of rk r rk l] by linarith
  next
    case Black
    then have rk r = 2 * bh r
      by (simp add: rk_def)
    moreover have rk (blacken ?t) = rk ?t
      using blacken_id[OF t(2)[THEN mp, OF Black]] by simp
    moreover have rk ?t ≤ 2 * bh r + 1
      using rk_t by simp
    ultimately show ?thesis
      using max.cobounded2[of rk r rk l] by linarith
  qed
qed

lemma rk_join:
  assumes rbt l rbt r
  shows max (rk l) (rk r) ≤ rk (join l x r) ∧ rk (join l x r) ≤ max (rk l) (rk r)
+ 1
proof −
  consider (R) bh r < bh l | (L) bh l < bh r | (EQ) bh l = bh r
    by linarith
  then show ?thesis
  proof cases
    case R
    then show ?thesis
      using rk_joinR[OF assms R] by simp
  next
    case L
    then show ?thesis
```

```
    using rk_joinL[OF assms L] by simp
  next
    case EQ
    then show ?thesis
      by (cases col l; cases col r; simp add: rk_def)
  qed
qed

lemmas rk_join_lower = rk_join[THEN conjunct1]
  and rk_join_upper = rk_join[THEN conjunct2]

corollary rule_mono:
  ⟦rbt l; rbt r⟧ ⟹ max (real (rk l)) (real (rk r)) ≤ real (rk (join l a r))
  using rk_join_lower by (metis of_nat_le_iff of_nat_max)

lemma rule_sub:
  assumes real (rk l') ≤ real (rk l) + x real (rk r') ≤ real (rk r) + x
  assumes rbt l' rbt r' rbt (Node l (a, b) r)
  shows real (rk (join l' a r')) ≤ real (rk (Node l (a, b) r)) + x
proof −
  obtain c h where b: b = (c, h)
    by (cases b)
  have 1: real (rk (join l' a r')) ≤ max (real (rk l')) (real (rk r')) + 1
    using rk_join_upper[OF assms(3,4)]
    by (metis of_nat_le_iff of_nat_max of_nat_1 of_nat_add)
  have 2: max (real (rk l')) (real (rk r')) ≤ max (real (rk l)) (real (rk r)) + x
    using assms(1,2) by (auto simp: max_def)
  have 3: max (real (rk l)) (real (rk r)) + 1 ≤ real (rk (Node l (a, b) r))
    using rk_children(1)[of l a c h r] assms(5) b
    by (simp flip: of_nat_max)
  show ?thesis
    using 1 2 3 by linarith
qed
```

11.4.3 Cost Rule

```
time_fun col
time_fun bh
time_fun R
time_fun B
time_fun baliL
time_fun baliR
time_fun blacken
time_fun joinL
time_fun joinR
time_fun join

lemma T_col_0[simp]: T_col t = 0
  by (cases t rule: tree2_cases) auto
```

```
lemma T_bh_0[simp]: T_bh t = 0
  by (cases t rule: tree2_cases) auto

lemma T_R_0[simp]: T_R l a r = 0 and T_B_0[simp]: T_B l a r = 0
  by simp_all

lemma T_baliL_0[simp]: T_baliL l a r = 0
  by (induction l a r rule: baliL.induct) auto

lemma T_baliR_0[simp]: T_baliR l a r = 0
  by (induction l a r rule: baliR.induct) auto

lemma T_blacken_0[simp]: T_blacken t = 0
  by (cases t rule: blacken.cases) auto

declare T_joinL.simps[simp del]
declare T_joinR.simps[simp del]

lemma T_joinL:
  assumes invc r invh r bh l ≤ bh r
  shows T_joinL l x r + 2 * bh l ≤ 2 * bh r + 1 + (if col r = Red then 1 else 0)
  using assms
proof (induction r rule: tree2_induct)
  case Leaf
  then show ?case
    by (simp add: T_joinL.simps)
next
  case (Node l' x' b r')
  obtain c h where b: b = (c, h)
    by (cases b)
  let ?r = Node l' (x', b) r'
  show ?case
  proof (cases bh ?r ≤ bh l)
    case True
    then show ?thesis
      using Node.prems(3) unfolding b
      by (simp add: T_joinL.simps[of l x Node l' (x', (c, h)) r'])
  next
    case False
    have T: T_joinL l x ?r = T_joinL l x l' + 1
      using False unfolding b
      by (simp add: T_joinL.simps[of l x Node l' (x', (c, h)) r'])
    have IH: T_joinL l x l' + 2 * bh l ≤ 2 * bh l' + 1 + (if col l' = Red then 1
else 0)
      using Node.IH(1) Node.prems False b by (auto split: if_splits)
    show ?thesis
      using IH T Node.prems b by (cases c; cases col l'; simp)
  qed
```

qed

lemma *T_joinR*:
assumes *invc l invh l bh r ≤ bh l*
shows *T_joinR l x r + 2 * bh r ≤ 2 * bh l + 1 + (if col l = Red then 1 else 0)*
using *assms*
proof (*induction l rule*: *tree2_induct*)
case *Leaf*
then show *?case*
by (*simp add*: *T_joinR.simps*)
next
case (*Node l' x' b r'*)
obtain *c h* **where** *b*: *b* = (*c*, *h*)
by (*cases b*)
let *?l* = *Node l'* (*x'*, *b*) *r'*
show *?case*
proof (*cases bh ?l ≤ bh r*)
case *True*
then show *?thesis*
using *Node.prems*(*3*) **unfolding** *b*
by (*simp add*: *T_joinR.simps*[*of Node l'* (*x'*, (*c*, *h*)) *r' x r*])
next
case *False*
have *T*: *T_joinR ?l x r* = *T_joinR r' x r* + *1*
using *False* **unfolding** *b*
by (*simp add*: *T_joinR.simps*[*of Node l'* (*x'*, (*c*, *h*)) *r' x r*])
have *IH*: *T_joinR r' x r* + *2 * bh r ≤ 2 * bh r'* + *1* + (*if col r'* = *Red then 1 else 0*)
using *Node.IH*(*2*) *Node.prems False b* **by** (*auto split*: *if_splits*)
show *?thesis*
using *IH T Node.prems b* **by** (*cases c*; *cases col r'*; *simp*)
qed
qed

lemma *rule_cost_R*:
assumes *rbt l rbt r bh r < bh l*
shows *T_join l x r + rk r ≤ rk l + 2*
proof −
have *T_joinR l x r + 2 * bh r ≤ 2 * bh l + 1 + (if col l = Red then 1 else 0)*
using *T_joinR*[*of l r x*] *assms* **by** *simp*
moreover have *T_join l x r* = *T_joinR l x r*
using *assms*(*3*) **by** *simp*
ultimately show *?thesis*
by (*cases col l*; *cases col r*; *simp add*: *rk_def*)
qed

lemma *rule_cost_L*:
assumes *rbt l rbt r bh l < bh r*
shows *T_join l x r + rk l ≤ rk r + 2*

proof −
have *T_joinL l x r + 2 * bh l ≤ 2 * bh r + 1 + (if col r = Red then 1 else 0)*
using *T_joinL[of r l x] assms* **by** *simp*
moreover have *T_join l x r = T_joinL l x r*
using *assms(3)* **by** *simp*
ultimately show *?thesis*
by (*cases col l*; *cases col r*; *simp add*: *rk_def*)
qed

lemma *rule_cost_EQ*: *bh l = bh r ⟹ T_join l x r = 0*
by *simp*

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

lemma *rule_cost*:
assumes *rbt l rbt r*
shows *real (T_join l x r) ≤ 2 + |real (rk l) − real (rk r)|*
proof −
consider (*R*) *bh r < bh l* | (*L*) *bh l < bh r* | (*EQ*) *bh l = bh r*
by *linarith*
then show *?thesis*
proof *cases*
case *R*
have *real (T_join l x r) + real (rk r) ≤ real (rk l) + 2*
using *rule_cost_R*[*OF assms R*, *of x*]
by (*metis of_nat_add of_nat_le_iff of_nat_numeral*)
then show *?thesis*
using *abs_ge_self*[*of real (rk l) − real (rk r)*] **by** *linarith*
next
case *L*
have *real (T_join l x r) + real (rk l) ≤ real (rk r) + 2*
using *rule_cost_L*[*OF assms L*, *of x*]
by (*metis of_nat_add of_nat_le_iff of_nat_numeral*)
then show *?thesis*
using *abs_ge_minus_self*[*of real (rk l) − real (rk r)*] **by** *linarith*
next
case *EQ*
then show *?thesis*
by (*simp add*: *rule_cost_EQ*)
qed
qed

11.5 Instantiation of *StronglyJoinable*

interpretation *tree_rb*: *StronglyJoinable*
where *join = join* **and** *inv = rbt*
and *rank = λt. real (rk t)* **and** $c_l = 1$ **and** $c_u = 2$
and *T_join = T_join* **and** $k_0 = 2$ **and** $k_1 = 1$
proof (*standard*, *goal_cases*)

```
  case (1 l r a)
  then show ?case
    by (rule rule_mono)
next
  case (2 l' l x r' r a b)
  then show ?case
    by (intro rule_sub)
next
  case (3 l a r)
  show ?case
    by (rule join_size)
next
  case (4 l r a)
  then show ?case
    using rule_cost[of l r a] by simp
next
  case 5
  then show ?case by simp
next
  case 6
  then show ?case by simp
qed
```

11.6 Concrete constants for red-black trees

```
lemma rbt_c_delta: tree_rb.c_δ = 1
  unfolding tree_rb.c_δ_def by simp

lemma rbt_C: tree_rb.C = 1
  unfolding tree_rb.C_def by simp

lemma rbt_K_split: tree_rb.K_split = 8
  unfolding tree_rb.K_split_def by simp

lemma rbt_K_T: tree_rb.K_T = 6
  unfolding tree_rb.K_T_def by simp

lemma rbt_K_min: tree_rb.K_min = 4
  unfolding tree_rb.c_1_def rbt_c_delta tree_rb.K_min_def by auto

lemma rbt_K_lin_union: tree_rb.K_T * tree_rb.K_lin 2 3 ≤ 1856
proof −
  have tree_rb.K_T * tree_rb.K_lin 2 3 = 1134 + 510 * sqrt 2
    unfolding tree_rb.K_lin_def
    by (simp add: rbt_K_T rbt_C rbt_K_split S1_two S2_two algebra_simps)
  also have ... ≤ 1134 + 510 * 1.415
    using sqrt2_bounds(3) by (auto intro: add_left_mono mult_left_mono)
  also have 1134 + 510 * (1.415::real) ≤ 1856
    by simp
```

```
  finally show ?thesis .
qed

lemma rbt_K_log_union: tree_rb.K_T * tree_rb.K_log 2 3 ≤ 943
proof −
  have tree_rb.K_T * tree_rb.K_log 2 3 = 756 + 132 * sqrt 2
    unfolding tree_rb.K_log_def
    by (simp add: rbt_K_T rbt_C rbt_K_split S1_two algebra_simps)
  also have ... ≤ 756 + 132 * 1.415
    using sqrt2_bounds(3) by (auto intro: add_left_mono mult_left_mono)
  also have 756 + 132 * (1.415::real) ≤ 943
    by simp
  finally show ?thesis .
qed

lemma rbt_K_lin_inter: tree_rb.K_T * tree_rb.K_lin 16 12 ≤ 3333
proof −
  have tree_rb.K_T * tree_rb.K_lin 16 12 = 2034 + 918 * sqrt 2
    unfolding tree_rb.K_lin_def
    by (simp add: rbt_K_T rbt_C rbt_K_split S1_two S2_two algebra_simps)
  also have ... ≤ 2034 + 918 * 1.415
    using sqrt2_bounds(3) by (auto intro: add_left_mono mult_left_mono)
  also have 2034 + 918 * (1.415::real) ≤ 3333
    by simp
  finally show ?thesis .
qed

lemma rbt_K_log_inter: tree_rb.K_T * tree_rb.K_log 16 12 ≤ 1696
proof −
  have tree_rb.K_T * tree_rb.K_log 16 12 = 1356 + 240 * sqrt 2
    unfolding tree_rb.K_log_def
    by (simp add: rbt_K_T rbt_C rbt_K_split S1_two algebra_simps)
  also have ... ≤ 1356 + 240 * 1.415
    using sqrt2_bounds(3) by (auto intro: add_left_mono mult_left_mono)
  also have 1356 + 240 * (1.415::real) ≤ 1696
    by simp
  finally show ?thesis .
qed

corollary rbt_T_union_linlog:
  assumes rbt t1 rbt t2
  shows real (tree_rb.T_union t1 t2)
      ≤ 6 + 1856 * size_min t1 t2
        + 943 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have real (tree_rb.T_union t1 t2)
     ≤ tree_rb.K_T + (tree_rb.K_T * tree_rb.K_lin 2 3) * size_min t1 t2
       + (tree_rb.K_T * tree_rb.K_log 2 3)
         * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
```

```
    using tree_rb.T_union_bound[OF assms]
    by (simp add: rbt_c_delta)
  also have ... ≤ 6 + 1856 * size_min t1 t2
      + 943 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using rbt_K_lin_union rbt_K_log_union rbt_K_T
    by (intro add_mono mult_right_mono) (auto simp: log_def)
  finally show ?thesis .
qed

corollary rbt_T_inter_linlog:
  assumes rbt t1 rbt t2
  shows real (tree_rb.T_inter t1 t2)
      ≤ 6 + 3333 * size_min t1 t2
        + 1696 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have real (tree_rb.T_inter t1 t2)
     ≤ tree_rb.K_T + (tree_rb.K_T * tree_rb.K_lin 16 12) * size_min t1 t2
       + (tree_rb.K_T * tree_rb.K_log 16 12)
         * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using tree_rb.T_inter_bound[OF assms]
    by (simp add: rbt_K_min)
  also have ... ≤ 6 + 3333 * size_min t1 t2
      + 1696 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using rbt_K_lin_inter rbt_K_log_inter rbt_K_T
    by (intro add_mono mult_right_mono) (auto simp: log_def)
  finally show ?thesis .
qed

corollary rbt_T_diff_linlog:
  assumes rbt t1 rbt t2
  shows real (tree_rb.T_diff t1 t2)
      ≤ 6 + 3333 * size_min t1 t2
        + 1696 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
proof −
  have real (tree_rb.T_diff t1 t2)
     ≤ tree_rb.K_T + (tree_rb.K_T * tree_rb.K_lin 16 12) * size_min t1 t2
       + (tree_rb.K_T * tree_rb.K_log 16 12)
         * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using tree_rb.T_diff_bound[OF assms]
    by (simp add: rbt_K_min)
  also have ... ≤ 6 + 3333 * size_min t1 t2
      + 1696 * (size_min t1 t2 * log 2 (size_max t1 t2 / size_min t1 t2 + 1))
    using rbt_K_lin_inter rbt_K_log_inter rbt_K_T
    by (intro add_mono mult_right_mono) (auto simp: log_def)
  finally show ?thesis .
qed

corollary rbt_T_union_bigo:
  (λ(t1, t2). real (tree_rb.T_union t1 t2)) ∈ O[tree_rb.tree_pair_filter](λ(t1, t2).
```

size_min t1 t2 ∗ ln (size_max t1 t2 / size_min t1 t2 + 1))
using *tree_rb.T_union_bigo* **.**

corollary *rbt_T_inter_bigo*:
(λ(*t1*, *t2*). *real* (*tree_rb.T_inter t1 t2*)) ∈ *O*[*tree_rb.tree_pair_filter*](λ(*t1*, *t2*).
size_min t1 t2 ∗ ln (size_max t1 t2 / size_min t1 t2 + 1))
using *tree_rb.T_inter_bigo* **.**

corollary *rbt_T_diff_bigo*:
(λ(*t1*, *t2*). *real* (*tree_rb.T_diff t1 t2*)) ∈ *O*[*tree_rb.tree_pair_filter*](λ(*t1*, *t2*).
size_min t1 t2 ∗ ln (size_max t1 t2 / size_min t1 t2 + 1))
using *tree_rb.T_diff_bigo* **.**

end

A Appendix

theory *Submodularity*
imports
StronglyJoinableRBT
begin

[4] states the submodularity rule in two parts, a decreasing and an increasing side. The locale below states both verbatim as *rule_dec* and *rule_inc*; in particular the increasing side bounds the rank of the join from below as well as from above.

locale *PaperJoinable* =
Set2_Join join inv +
BalancedTree rank c_l c_u *inv*
for *rank* :: (*'a*::*linorder* ∗ *'b*) *tree* ⇒ *real*
and c_l c_u :: *real*
and *join* :: (*'a*∗*'b*) *tree* ⇒ *'a* ⇒ (*'a*∗*'b*) *tree* ⇒ (*'a*∗*'b*) *tree*
and *inv* :: (*'a*∗*'b*) *tree* ⇒ *bool*
+
assumes *join_size*: ⟦*inv l*; *inv r*⟧ ⟹ *size* (*join l a r*) = *size l* + *size r* + *1*
assumes *rule_dec*:
⟦ *rank l′* ≤ *rank l*; *rank r′* ≤ *rank r*; *inv l′*; *inv r′*; *inv* (*Node l* (*a*,*b*) *r*) ⟧ ⟹
rank (*join l′ a r′*) ≤ *rank* (*Node l* (*a*,*b*) *r*)
assumes *rule_inc*:
⟦ *rank l* ≤ *rank l′*; *rank l′* ≤ *rank l* + *x*; *rank r* ≤ *rank r′*; *rank r′* ≤ *rank r* +
x;
inv l′; *inv r′*; *inv* (*Node l* (*a*,*b*) *r*) ⟧ ⟹
rank (*Node l* (*a*,*b*) *r*) ≤ *rank* (*join l′ a r′*) ∧
rank (*join l′ a r′*) ≤ *rank* (*Node l* (*a*,*b*) *r*) + *x*
begin
end

The main locale recovers the decreasing side verbatim and the upper bound of the increasing side.

```
lemma (in StronglyJoinable) rule_sub_inc_upper:
  ⟦ rank l ≤ rank l'; rank l' ≤ rank l + x; rank r ≤ rank r'; rank r' ≤ rank r + x;
     inv l'; inv r'; inv (Node l (a,b) r) ⟧ ⟹
     rank (join l' a r') ≤ rank (Node l (a,b) r) + x
  by (blast intro: rule_sub)
```

The lower bound of the increasing side is not recovered, and it cannot be. Red-black trees are an instance of *StronglyJoinable*, but over red-black trees the two bounds of the increasing rule contradict each other already at a singleton, for any *join* of the type fixed by *Set2_Join*.

Conversely, the rules of [4] do not imply *rule_sub* either, which is stated for arbitrary real x and thus also covers mixed and strictly decreasing rank changes. A counterexample can be constructed but is outside the scope of this formalization.

```
lemma no_join_satisfies_paper_inc_rule:
  fixes J :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
  assumes inc:
    ⋀l l' r r' a b x.
      ⟦ real (rk l) ≤ real (rk l'); real (rk l') ≤ real (rk l) + x;
        real (rk r) ≤ real (rk r'); real (rk r') ≤ real (rk r) + x;
        rbt l'; rbt r'; rbt (Node l (a,b) r) ⟧ ⟹
      real (rk (Node l (a,b) r)) ≤ real (rk (J l' a r')) ∧
      real (rk (J l' a r')) ≤ real (rk (Node l (a,b) r)) + x
  shows False
proof −
  let ?t_red = Node Leaf (a::'a, (Red, 0)) Leaf
  let ?t_black = Node Leaf (a, (Black, Suc 0)) Leaf
  let ?join = J Leaf a Leaf

  have rbt ?t_red and rbt ?t_black
    by auto

  have real (rk ?t_red) ≤ real (rk ?join) ∧
       real (rk ?join) ≤ real (rk ?t_red) + 0
    by (intro inc) auto
  then have upper: real (rk ?join) ≤ 1
    by simp

  have real (rk ?t_black) ≤ real (rk ?join) ∧
       real (rk ?join) ≤ real (rk ?t_black) + 0
    by (intro inc) auto
  then have lower: 2 ≤ real (rk ?join)
    by simp

  from upper lower show False
    by linarith
qed
```

Stated against the locale; no *join* whatsoever makes red-black trees an

instance of the rules of [4].

```
corollary no_PaperJoinable_rbt:
  fixes J :: ('a::linorder) rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
  shows ¬ PaperJoinable (λt. real (rk t)) 1 2 J rbt
proof
  assume A: PaperJoinable (λt. real (rk t)) 1 2 J rbt
  show False
        by (rule no_join_satisfies_paper_inc_rule[of J]) (rule PaperJoin-
able.rule_inc[OF A])
qed
```

The flag of the paper resolves the contradiction. Ported to this setting, the flagged join receives the colour of the node being rebuilt and consults it in the case of equal black heights, where a flag-free *join* has to commit to one colour.

```
fun join_f :: color ⇒ 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where
join_f c l x r =
  (if bh r < bh l then blacken (joinR l x r)
   else if bh l < bh r then blacken (joinL l x r)
   else if c = Red ∧ col l = Black ∧ col r = Black then R l x r else B l x r)

corollary join_f_rebuilds:
  assumes rbt (Node l (a, (c, h)) r)
  shows join_f c l a r = Node l (a, (c, h)) r
using assms by auto
```

In particular the two anchors of *no_join_satisfies_paper_inc_rule* now constrain two distinct calls, and both are satisfied exactly.

```
corollary join_f_resolves_singletons:
  join_f Red Leaf a Leaf = Node Leaf (a, (Red, 0)) Leaf
  join_f Black Leaf a Leaf = Node Leaf (a, (Black, Suc 0)) Leaf
  by auto
```

The author concedes, that this appendix does not represent a comprehensive argument for stating the submodularity rule(s) one way or another. In particular, it might be possible to define *rk* or the *rbt* variant in a way that resolves the problem at hand, but this direction was not explored thoroughly. Rather, the aim is to provide an argument for why the submodularity rule was modified and why the author considers it sound to have done so.

```
end
```

References

[1] S. Adams. Functional pearls: Efficient sets – a balancing act. *Journal of Functional Programming*, 3(4):553–561, 1993.

[2] S. Adams. MIT/GNU Scheme release 12.1, runtime: weight-balanced trees (wttree.scm). https://git.savannah.gnu.org/cgit/mit-scheme.git/tree/src/runtime/wttree.scm?h=release-12.1, 2023. Code written 1993–1994. Accessed 2026-09-09.

[3] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962. English translation of Doklady Akademii Nauk SSSR 146(2):263–266.

[4] G. E. Blelloch, D. Ferizovic, and Y. Sun. Joinable parallel balanced binary trees. *ACM Transactions on Parallel Computing*, 9(2):7:1–7:41, 2022.

[5] J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, and S. Weirich. Ready, set, verify! Applying hs-to-coq to real-world Haskell code (experience report). *Proceedings of the ACM on Programming Languages*, 2(ICFP):89:1–89:16, 2018.

[6] EPFL and Lightbend. Scala 2.13.18 standard library: scala.collection.immutable.RedBlackTree. https://github.com/scala/scala/blob/v2.13.18/src/library/scala/collection/immutable/RedBlackTree.scala, 2025. Accessed 2026-09-09.

[7] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (SFCS 1978)*, pages 8–21. IEEE, 1978.

[8] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[9] Jane Street. Base, version 0.17.3: module Set. https://github.com/janestreet/base/blob/v0.17.3/src/set.ml, 2025. Accessed 2026-09-09.

[10] D. Leijen et al. containers, version 0.8: module Data.Set.Internal. https://hackage.haskell.org/package/containers-0.8/docs/src/Data.Set.Internal.html, 2025. Haskell package. Accessed 2026-09-09.

[11] X. Leroy et al. The OCaml system, release 5.5.1: standard library module Set. https://github.com/ocaml/ocaml/blob/5.5.1/stdlib/set.ml, 2026. Accessed 2026-09-09.

[12] R. Li, H. Grodin, and R. Harper. A verified cost analysis of joinable red-black trees, 2023. arXiv:2309.11056, https://doi.org/10.48550/arXiv.2309.11056.

[13] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[14] T. Nipkow, editor. *Functional Data Structures and Algorithms: A Proof Assistant Approach*. ACM Books, 2025. Online version at https://fdsa-book.net/.

[15] J. Reppy and S. Adams. SML/NJ 2026.2, SML/NJ library: functor BinarySetFn. https://github.com/smlnj/smlnj/blob/v2026.2/smlnj-lib/Util/binary-set-fn.sml, 2026. Adapted from Adams' 1992 implementation. Accessed 2026-09-09.

[16] Y. Sun, G. E. Blelloch, and D. Ferizovic. PAM: Parallel augmented maps, release V1. https://github.com/cmuparlay/PAM/tree/V1, 2021. Accessed 2026-09-16.

[17] Y. Sun, D. Ferizovic, and G. E. Blelloch. PAM: Parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*, pages 290–304. ACM, 2018.

[18] The HOL4 developers. HOL4 trindemossen-2, examples: theory balanced_map. https://github.com/HOL-Theorem-Prover/HOL/blob/trindemossen-2/examples/balanced_bst/balanced_mapScript.sml, 2025. Ported from `Data.Map.Base` of containers (GHC 7.8.3). Accessed 2026-09-09.

[19] The Rocq Development Team. The rocq standard library, version 9.2.0: module MSetAVL. https://github.com/rocq-prover/stdlib/blob/V9.2.0/theories/MSets/MSetAVL.v, 2026. Accessed 2026-09-09.