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*⟩

lemma *log_2powr_plus_1_le*:
fixes x :: *real*
assumes $x \geq 0$
shows *log 2 (2 powr x + 1)* $\leq 1 + x$
⟨*proof*⟩

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

corollary *ln_diff_less*: $0 < x \Longrightarrow 0 < y \Longrightarrow x \neq y \Longrightarrow$ *ln x − ln y* $< (x - y)$ / *y* **for**
x :: *real*
⟨*proof*⟩

lemma *ln_mono_1*:
fixes n :: *real*
assumes *n_pos*: $n > 0$
defines $f \equiv (\lambda x{::}real.\ x * ln\ (1 + n/x))$
shows *strict_mono_on* {*0*<..} *f*
⟨*proof*⟩

corollary *log_mono_1*:
fixes n :: *real*
assumes *n_pos*: $n > 0$
defines $f \equiv (\lambda x{::}real.\ x * log\ 2\ (1 + n/x))$
shows *strict_mono_on* {*0*<..} *f*
⟨*proof*⟩

corollary *xlog_mono*:
fixes $n\ x\ z$ :: *real*
assumes $n > 0\ 0 < x\ x \leq z$
shows $x * log\ 2\ (n\ /\ x + 1) \leq z * log\ 2\ (n\ /\ z + 1)$
⟨*proof*⟩

corollary *xlog_term_mono*:
fixes $n\ x\ z\ k\ c$ :: *real*
assumes $0 < x\ x \leq z\ 0 < n\ 0 \leq k\ 0 \leq c$
shows $x * (k + c * log\ 2\ (n\ /\ x + 1)) \leq z * (k + c * log\ 2\ (n\ /\ z + 1))$
⟨*proof*⟩

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 $(\lambda n.\ of\_nat\ n * z\ \hat{}\ n)\ sums\ (z\ /\ (1 - z)^2)$

⟨*proof*⟩

corollary *arith_geometric_summable*:
fixes *c* :: *real*
assumes *norm c < 1*
shows *summable* (λ*n*. *n* ∗ *c*^*n*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *geom_ratio_arith_sum_le*:
assumes *cu > 0 finite A*
shows (∑ *i*∈*A*. *real i* ∗ (*geom_ratio cu*) ^ *i*) ≤ *S2 cu*
⟨*proof*⟩

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*⟩

lemma *S1_two*: *S1 2 = 2 + sqrt 2*
⟨*proof*⟩

lemma *S2_two*: *S2 2 = 4 + 3 * sqrt 2*
⟨*proof*⟩

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* ≡ (λ*i*::*'b*. *1 / card S*)
  assumes *finite S S* ≠ {}
    and *concave_on D f*
    and ⋀*i*. *i* ∈ *S* ⟹ *x i* ∈ *D*
  shows *f* ((*1 / card S*) *$*_R$ (∑ *i*∈*S*. *x i*))
    ≥ (*1 / card S*) * (∑ *i*∈*S*. *f* (*x i*))
⟨*proof*⟩

corollary *jensen_list*:
  assumes *xs* ≠ []
    and *concave_on C f*
    and ⋀*x*. *x* ∈ *set xs* ⟹ *x* ∈ *C*
  shows *f* ((*1 / length xs*) *$*_R$ *sum_list xs*)
    ≥ (*1 / length xs*) * *sum_list* (*map f xs*)
⟨*proof*⟩

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* $\Rightarrow$ *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* $< c_l$
⟨*proof*⟩

lemma *c_l_le_one*: $c_l \leq 1$
⟨*proof*⟩

lemma *c_u_ge_one*: *1* $\leq c_u$
⟨*proof*⟩

corollary *c_l_nonneg*: *0* $\leq c_l$
⟨*proof*⟩

corollary *c_u_pos*: *0* $< c_u$
⟨*proof*⟩

corollary *c_u_nonneg*: *0* $\leq c_u$
⟨*proof*⟩

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*$| \leq c_\delta$
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

corollary *rank_lower_height_inverse*:*balanced t* $\Longrightarrow$ *height t* $\leq$ *rank t* / c_l
⟨*proof*⟩

Which also implies *rank* is never negative.

corollary *rank_pos*:*balanced t* $\Longrightarrow$ *rank t* $\geq$ *0*
⟨*proof*⟩

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*⟩

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*⟩

lemma *size1_ge_powr_rank*:
balanced t $\Longrightarrow$ *size1 t* $\geq$ *2 powr* (*rank t* / c_u)
⟨*proof*⟩

corollary *rank_le_cu_log_size1*:
assumes *balanced t*
shows *rank t* $\leq$ c_u $*$ *log 2* (*size1 t*)
⟨*proof*⟩

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* = []
⟨*proof*⟩

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*)
⟨*proof*⟩

Balance is inherited.

lemma *balanced_rank_roots*: ⟦*balanced t*; *r* ∈ *set*(*rank_roots t i*)⟧ ⟹ *balanced r*
⟨*proof*⟩

lemma *layer_eq_concat_rank_roots*:
layer T i = *concat* (*map* (λ*r*. *layer r i*) (*rank_roots T i*))
⟨*proof*⟩

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*⟩

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

lemma *rank_roots_size_sum*:
$sum\_list\ (map\ size1\ (rank\_roots\ t\ i)) \leq size1\ t$
⟨*proof*⟩

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*⟩

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*⟩

A balanced tree whose rank overshoots layer i by at most d has at most $2^{\left\lceil \frac{d}{c_l} \right\rceil} - 1$ 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 $0 \leq d$
shows $length\ (layer\ t\ i) \leq 2\hat{}(nat\ (ceiling\ (d\ /\ c_l))) - 1$
⟨*proof*⟩

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

definition $C :: nat$ **where** $C = 2\hat{}(nat\ (ceiling\ (1\ /\ c_l))) - 1$

lemma *C_pos*: $real\ C \geq 0$
⟨*proof*⟩

corollary *layer_bound_const*:
assumes $balanced\ t\ rank\ t < real\ i + 1$
shows $length\ (layer\ t\ i) \leq C$
⟨*proof*⟩

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*:
$length\ (concat\ (map\ f\ xs)) \leq sum\_list\ (map\ (\lambda x.\ length\ (f\ x))\ xs)$
⟨*proof*⟩

lemma *layer_bound_via_rank_roots*:
assumes *BT*: *balanced T*
shows *length* (*layer T i*)
$\leq C * length\ (rank\_roots\ T\ i)$
⟨*proof*⟩

Together, this yields the fully combined geometric layer bound.

corollary *layer_bound_geometric*:
assumes *balanced t*
shows $(length\ (layer\ t\ i)) \leq C * (size1\ t\ /\ (2\ powr\ (i\ /\ c_u)))$
⟨*proof*⟩

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* :: $('a \times 'b)\ tree \Rightarrow real$
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*]: $inv\ t \Longrightarrow 0 \leq rank\ t$
⟨*proof*⟩

corollary *dmax'*: *inv* (*Node l a r*) $\Longrightarrow$ |*rank l* − *rank r*| ≤ c_δ
⟨*proof*⟩

corollary *rank_le_cu_log_size1'*: *inv t* $\Longrightarrow$ *rank t* ≤ c_u ∗ *log 2* (*size1 t*)
⟨*proof*⟩

corollary *rank_lower_height_inverse'*: *inv t* $\Longrightarrow$ *height t* ≤ *rank t* / c_l
⟨*proof*⟩

corollary *inv_children_explD*:
assumes *inv* (*Node l a r*)
shows *rank l* ≤ *rank* (*Node l a r*) − c_l
and *rank r* ≤ *rank* (*Node l a r*) − c_l
and *rank* (*Node l a r*) − c_u ≤ *rank l*
and *rank* (*Node l a r*) − c_u ≤ *rank r*
⟨*proof*⟩
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*
  ⟨*proof*⟩

lemma *split_invR*[*dest*]: ⟦*split x t* = (*l*,*b*,*r*); *inv t*⟧ ⟹ *inv r*
  ⟨*proof*⟩

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

⟨*proof*⟩

lemma *split_invR_sym*[*dest*]: ⟦(*l*,*b*,*r*) = *split x t*; *inv t*⟧ ⟹ *inv r*
⟨*proof*⟩

lemma *obtain_inv*[*elim*]:
assumes *inv* (*Node l* (*x*,*y*) *r*)
obtains *inv l inv r*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *join_rank_upper*:
assumes *inv l inv r*
shows *rank* (*join l a r*) ≤ *max* (*rank l*) (*rank r*) + c_u
⟨*proof*⟩

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* ⇒ '*a* ⇒ ('*a* ∗ '*b*) *tree* ⇒ *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* ≤ *K_T*
⟨*proof*⟩

lemma *K_T_ge_k1*: k_1 ≤ *K_T*
⟨*proof*⟩

lemma *K_T_ge_1_plus_k0*: $1 + k_0 \leq K\_T$
⟨*proof*⟩

lemma *W_join_nonneg*[*simp*]: $0 \leq W\_join\ l\ a\ r$
⟨*proof*⟩

lemma *rule_cost_W*:
$[\![inv\ l;\ inv\ r]\!] \Longrightarrow real\ (T\_join\ l\ a\ r) \leq k_0 + k_1 * W\_join\ l\ a\ r$
⟨*proof*⟩

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 ⇒ splitl l x |
EQ ⇒ l |
GT ⇒ join l a (splitl r x))

fun *splitr* :: $('a * 'b)\ tree \Rightarrow 'a \Rightarrow ('a * 'b)\ tree$ **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*
⟨*proof*⟩

lemma *splitr_eq_r*:*split x t = (l,b,r) ⟹ splitr t x = r*
⟨*proof*⟩

corollary *splitl_fst*: *splitl t x = fst (split x t)*
⟨*proof*⟩

corollary *splitr_snd*: *splitr t x = snd (snd (split x t))*
⟨*proof*⟩

corollary *inv_splitl*:*inv t ⟹ inv (splitl t x)*
⟨*proof*⟩

corollary *inv_splitr*:*inv t ⟹ inv (splitr t x)*
⟨*proof*⟩

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* $\Longrightarrow$ *rank t* $\geq$ *rank* (*splitl t x*)
⟨*proof*⟩

lemma *splitr_rank*:*inv t* $\Longrightarrow$ *rank t* $\geq$ *rank* (*splitr t x*)
⟨*proof*⟩

corollary *split_rank*: ⟦*split x t* = (*l*,*b*,*r*); *inv t*⟧ $\Longrightarrow$ *rank l* $\leq$ *rank t* $\wedge$ *rank r* $\leq$ *rank t*
⟨*proof*⟩

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* $\neq$ *Leaf split x t* = (*l*, *b*, *r*)
shows *min* (*rank l*) (*rank r*) $\leq$ *rank t* − c_l
⟨*proof*⟩

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*⟧ $\Longrightarrow$
inv l $\wedge$ *inv r* $\wedge$ *size t* = *size l* + *size r* + (*if b then 1 else 0*)
⟨*proof*⟩

corollary *split_size*: ⟦(*l*, *b*, *r*) = *split x t*; *inv t*⟧ $\Longrightarrow$ *size l* + *size r* $\leq$ *size t*
⟨*proof*⟩

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*⟧ $\Longrightarrow$ *size l* + *size r* = *size t* − *1*
⟨*proof*⟩

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
  ⟨proof⟩
```

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 c1 = 2 * cδ + 1

lemma c1_pos: c1 > 0
  ⟨proof⟩

lemma W_splitl_bounded:
assumes inv t
shows W_splitl t x ≤ 1 + c1 * height t + rank (splitl t x)
⟨proof⟩
```

The right side is mirrored.

```
lemma W_splitr_bounded:
assumes inv t
shows W_splitr t x ≤ 1 + c1 * height t + rank (splitr t x)
⟨proof⟩
```

Combining both yields the work bound of *split*.

```
definition K_split where
K_split = (4 * cu − 2 * cl + 2) / cl
```

lemma *K_split_pos*: $K\_split > 0$
⟨*proof*⟩

lemma *K_split_ge_1*: $K\_split \geq 1$
⟨*proof*⟩

theorem *W_split_bounded*:
assumes *inv t*
shows $W\_split\ t\ x \leq 2 + K\_split * rank\ t$
⟨*proof*⟩

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*: $1 \leq W\_split\ t\ x$
⟨*proof*⟩

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*⟩

lemma *T_split_bridge_strong*:
$inv\ t \Longrightarrow real\ (T\_split\ x\ t) \leq (1 + k_0 + k_1) * W\_split\ t\ x$
⟨*proof*⟩

corollary *T_split_bridge*:
assumes *inv t*
shows $real\ (T\_split\ x\ t) \leq K\_T * W\_split\ t\ x$
⟨*proof*⟩

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') $t \neq Leaf$
shows $rank\ t \geq rank\ t'$
⟨*proof*⟩

lemma *split_min_rank_min*:
assumes *inv t split_min t* = (m, t') $t \neq Leaf$
shows $rank\ t \leq rank\ t' + c_u$
⟨*proof*⟩

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*
⟨*proof*⟩

corollary *split_min_size*:
assumes *inv t split_min t* = (*m*, *t'*) *t* ≠ *Leaf*
shows *size t'* = *size t* − *1*
⟨*proof*⟩

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'*
⟨*proof*⟩

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

lemma *K_min_nonneg*: *0* ≤ *K_min*
⟨*proof*⟩

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*⟩

lemma *W_split_min_ge_1*: *t* ≠ *Leaf* ⟹ *1* ≤ *W_split_min t*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *T_split_min_bridge_strong*:
⟦*inv t*; *t* ≠ *Leaf*⟧ ⟹ *real* (*T_split_min t*) ≤ (*1* + k_0 + k_1) ∗ *W_split_min t*
⟨*proof*⟩

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*: $[\![split\_min\ r = (m,\ r');\ r \neq Leaf]\!] \Longrightarrow join2\ l\ r = join\ l\ m\ r'$
⟨*proof*⟩

lemma *rule_mono_join2*:
assumes *inv l inv r*
shows $max\ (rank\ l)\ (rank\ r) \leq rank\ (join2\ l\ r) + c_u$
⟨*proof*⟩

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*⟩

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*:
$[\![split\_min\ r = (m,\ r');\ r \neq Leaf]\!] \Longrightarrow$
W_join2 l r = *W_join l m r'* + *W_split_min r*
⟨*proof*⟩

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*⟩

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* ≤ (1 + k_0 + k_1) ∗ *w* **and** 1 ≤ *w*
shows *real t* + *real* (*T_join l a r*) ≤ *K_T* ∗ (*W_join l a r* + *w*)
⟨*proof*⟩

lemma *T_join2_bridge*:
assumes *inv l inv r*
shows *real* (*T_join2 l r*) ≤ *K_T* ∗ *W_join2 l r*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *split_parts_inv*:
assumes *inv t2*
shows *t* ∈ *set* (*split_parts t1 t2*) ⟹ *inv t*
⟨*proof*⟩

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

```
fun parts_in_layer :: (('a::linorder) ∗ 'b) tree ⇒ ('a ∗ 'b) tree ⇒ nat ⇒ ('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 ⇒
       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* ≤ *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* = []
⟨*proof*⟩

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*)

$= (\sum i \leq nat\ \lceil rank\ t1 \rceil.\ \sum t{\leftarrow}parts\_in\_layer\ t1\ t2\ i.\ f\ t)$
⟨*proof*⟩

A band condition $real\ j \leq x \wedge x < real\ j + 1$ 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 $0 \leq x$ **and** $nat\ \lfloor x \rfloor \leq R$
shows $(\sum j \leq R.\ (if\ in\_band\ j\ x\ then\ c\ else\ 0)) = c$
⟨*proof*⟩

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 $(\sum t{\leftarrow}split\_parts\ t1\ t2.\ f\ t) = (\sum i \leq nat\ \lceil rank\ t1 \rceil.\ \sum t{\leftarrow}parts\_in\_layer\ t1\ t2\ i.\ f\ t)$
⟨*proof*⟩

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* :: $(('a{::}linorder) * 'b)\ tree \Rightarrow ('a * 'b)\ tree \Rightarrow nat \Rightarrow ('a * 'b)\ tree\ list$ **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 $t \in set\ (rank\_root\_parts\_in\_layer\ t1\ t2\ i) \Longrightarrow inv\ t$
⟨*proof*⟩

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

lemma *root_parts_nonempty*:
$t \in set\ (rank\_root\_parts\_in\_layer\ t1\ t2\ i) \Longrightarrow t \neq Leaf$
⟨*proof*⟩

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 $(\sum t \leftarrow rank\_root\_parts\_in\_layer\ t1\ t2\ i.\ size\ t) \leq size\ t2$
⟨*proof*⟩

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*:
$length\ (rank\_root\_parts\_in\_layer\ t1\ t2\ i) \leq length\ (rank\_roots\ t1\ i)$
⟨*proof*⟩

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

lemma *part_rank_le_snd*:
$t \in set\ (parts\_in\_layer\ t1\ t2\ i) \Longrightarrow inv\ t2 \Longrightarrow rank\ t \leq rank\ t2$
⟨*proof*⟩

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 $length\ (parts\_in\_layer\ t1\ t2\ i) \leq C$
⟨*proof*⟩

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*⟩

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*$)$
⟨*proof*⟩

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 $(\sum t \leftarrow$ *split_parts t1 t2. k + c * rank t*$)$
    $\leq$ *real C* $*$ $(\sum i \leq$ *nat* $\lceil$*rank t1*$\rceil$.
      $\sum t \leftarrow$ *rank_root_parts_in_layer t1 t2 i. k + c * rank t*$)$
⟨*proof*⟩

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*⟩

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*$)$
  ⟨*proof*⟩

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*$)$
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*))
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)))
⟨*proof*⟩

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 $(\sum i \in L\ t1\ t2.$
$(size1\ t1\ /\ 2\ powr\ (i\ /\ c_u)) *$
$(k + c * c_u * log\ 2\ (size\ t2\ /\ (size1\ t1\ /\ 2\ powr\ (i\ /\ c_u)) + 1)))$
$\leq size1\ t1 * (k * S1\ c_u + c * c_u * S1\ c_u + c * S2\ c_u)$
$+ c * c_u * S1\ c_u * size1\ t1 * log\ 2\ (size\ t2\ /\ size1\ t1 + 1)$
⟨*proof*⟩

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 ≤ k* **and** *c0*: *0 ≤ 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*⟩

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 ≤ S1 c_u* **and** *S2_nonneg*: *0 ≤ S2 c_u*
⟨*proof*⟩

abbreviation *bound_mn* :: *real ⇒ real ⇒* (*'a∗'b*) *tree ⇒* (*'a∗'b*) *tree ⇒ real* **where**
bound_mn k c ta tb ≡
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 ≤ k* **and** *c0*: *0 ≤ c*
shows $(\sum i \in L\ t1\ t2.\ \sum t{\leftarrow}rank\_root\_parts\_in\_layer\ t1\ t2\ i.\ k + c * rank\ t) \leq bound\_mn\ k\ c\ t1\ t2$
⟨*proof*⟩

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*⟩

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

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

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*
⟨*proof*⟩

lemma *K_log_nonneg*: *0 ≤ k ⟹ 0 ≤ c ⟹ 0 ≤ K_log k c*

⟨*proof*⟩

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* ≤ (∑ *t* ← *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*⟩

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*:
(∑ *t* ← *xs*. (*k* + *1*) + *c* ∗ *rank t*) = *length xs* + (∑ *t* ← *xs*. *k* + *c* ∗ *rank t*)
⟨*proof*⟩

corollary *W_bound_generic′*:
fixes *W jw k c* :: *real*
assumes *invs*: *inv ta inv tb*
assumes *decomp*: *W* ≤ *1* + *length* (*split_parts ta tb*) + *jw* + *split_work ta tb*
assumes *jw_piv*: *jw* ≤ (∑ *t* ← *split_parts ta tb*. *k* + *c* ∗ *rank t*)
assumes *k_pos*: *0* < *k* **and** *c_pos*: *0* < *c*
shows *W* ≤ *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*⟩

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*
⟨*proof*⟩

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*⟩

corollary *W_bound_absorb*:
  fixes *W jw k c* :: *real*
  assumes *invs*: *inv ta inv tb*
  assumes *decomp*: $W \leq 1 + \mathit{length}\ (\mathit{split\_parts}\ ta\ tb) + jw + \mathit{split\_work}\ ta\ tb$
  assumes *jw_piv*: $jw \leq (\sum t \leftarrow \mathit{split\_parts}\ ta\ tb.\ k + c * \mathit{rank}\ t)$
  assumes *k_pos*: $0 < k$ **and** *c_pos*: $0 < c$
  shows $W \leq (1 + K\_lin\ k\ c + K\_log\ k\ c)$
        $* (\mathit{size\_min}\ ta\ tb * \mathit{log}\ 2\ (\mathit{size\_max}\ ta\ tb\ /\ \mathit{size\_min}\ ta\ tb + 1))$
⟨*proof*⟩

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* :: $('a \times 'b)$ *tree filter* **where**
  tree_filter = *size going_to at_top within* $\{t.\ \mathit{inv}\ t\}$

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* $(\lambda t.\ \mathit{inv}\ t \wedge 1 \leq \mathit{size}\ t)$ *tree_filter*
⟨*proof*⟩

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 $\lambda t.\ \mathit{False}$, 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*: $\exists\, t.\ \mathit{inv}\ t \wedge \mathit{size}\ t = N$
⟨*proof*⟩

Properness is now immediate.

lemma *tree_filter_proper*: *tree_filter* $\neq$ *bot*
⟨*proof*⟩

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* :: $((('a \times 'b)\ \mathit{tree}) \times ('a \times 'b)\ \mathit{tree})\ \mathit{filter}$ **where**
  tree_pair_filter = *tree_filter* $\times_F$ *tree_filter*

corollary *tree_pair_filter_proper*: *tree_pair_filter* $\neq$ *bot*

⟨*proof*⟩

corollary *eventually_tree_pair_filter*:
eventually (λ*x*. (λ*t*. *inv t* ∧ *1* ≤ *size t*) (*fst x*) ∧ (λ*t*. *inv t* ∧ *1* ≤ *size t*) (*snd x*))
(*tree_filter* ×$_F$ *tree_filter*)
⟨*proof*⟩

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*⟩

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*⟩

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
  ⟨proof⟩
```

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⟩

lemma union_rank_lower: ⟦inv t1; inv t2⟧ ⟹ rank t1 ≤ rank (union t1 t2) +
(c_u / c_l) * rank t2
⟨proof⟩
```

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
  ⟨proof⟩
```

lemma *KJc_union_pos*: $0 < 1 + c_u \;/\; c_l$
⟨*proof*⟩

fun *join_work_union'* :: $('a*'b)tree \Rightarrow ('a*'b)tree \Rightarrow real$ **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* ≤ *join_work_union' t1 t2*
⟨*proof*⟩

lemma *join_work_union'_as_sum*:
join_work_union' t1 t2 = $(\sum t \leftarrow$ *split_parts t1 t2*. *KJ_union + KJc_union * rank t*)
⟨*proof*⟩

corollary *join_work_union_le_sum_rank*:
assumes *inv t1 inv t2*
shows *join_work_union t1 t2* ≤ $(\sum t \leftarrow$ *split_parts t1 t2*. *KJ_union + KJc_union * rank t*)
⟨*proof*⟩

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*⟩

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

lemma *rule_cost_step_union*:
assumes *inv l inv r*
and *real* $t_s \le K\_T * w_s$ *real* $t_l \le K\_T * w_l$ *real* $t_r \le K\_T * w_r$
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*⟩

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*⟩

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*⟩

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*⟩

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

lemma *W_split_nonneg*: *0* ≤ *W_split t x*
⟨*proof*⟩

lemma *W_union_nonneg*: *0* ≤ *W_union t1 t2*
⟨*proof*⟩

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*⟩

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*⟩

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
  ⟨proof⟩
```

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
⟨proof⟩
```

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⟩

corollary inter_rank_le_log_size:
```

assumes *inv t1 inv t2*
shows *rank* (*inter t1 t2*) $\leq c_u *$ *log 2* (*size1 t2*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cu_log_size1_mono*:
fixes *s t* :: (′*a* ∗ ′*b*) *tree*
assumes *size s* ≤ *size t*
shows $c_u *$ *log 2* (*size1 s*) $\leq c_u *$ *log 2* (*size1 t*)
⟨*proof*⟩

lemma *join_work_inter′_as_sum*:
assumes *inv t1 inv t2*
shows *join_work_inter′ t1 t2*
≤ ($\sum$ *t* ← *split_parts t1 t2*.
2 + c_u + (*2* + *K_min*) ∗ c_u ∗ *log 2* (*size1 t*))
⟨*proof*⟩

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*
≤ ($\sum$ *t* ← *split_parts t1 t2*. *KJk_inter* + *KJc_inter* ∗ *rank t*)
⟨*proof*⟩

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*⟩

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*⟩

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

lemma *T_inter_bridge*:
  $[\![inv\ t1; inv\ t2]\!] \Longrightarrow real\ (T\_inter\ t1\ t2) \leq K\_T * W\_inter\ t1\ t2$
⟨*proof*⟩

lemma *KJk_inter_pos*: $0 < KJk\_inter$
⟨*proof*⟩

lemma *KJc_inter_pos*: $0 < KJc\_inter$
  ⟨*proof*⟩

theorem *W_inter_bound*:
  assumes *inv t1 inv t2*
  shows *W_inter t1 t2*
      $\leq 1 + K\_lin\ KJk\_inter\ KJc\_inter * size\_min\ t1\ t2$
        $+ K\_log\ KJk\_inter\ KJc\_inter$
          $* (size\_min\ t1\ t2 * log\ 2\ (size\_max\ t1\ t2\ /\ size\_min\ t1\ t2 + 1))$
  ⟨*proof*⟩

corollary *T_inter_bound*:
  assumes *inv t1 inv t2*
  shows *real* $(T\_inter\ t1\ t2)$
      $\leq K\_T + (K\_T * K\_lin\ KJk\_inter\ KJc\_inter) * size\_min\ t1\ t2$
        $+ (K\_T * K\_log\ KJk\_inter\ KJc\_inter)$
          $* (size\_min\ t1\ t2 * log\ 2\ (size\_max\ t1\ t2\ /\ size\_min\ t1\ t2 + 1))$
⟨*proof*⟩

lemma *W_split_min_nonneg*: $t \neq Leaf \Longrightarrow 0 \leq W\_split\_min\ t$
  ⟨*proof*⟩

lemma *W_join2_nonneg*: $0 \leq W\_join2\ l\ r$
  ⟨*proof*⟩

lemma *W_inter_nonneg*: *0 ≤ W_inter t1 t2*
⟨*proof*⟩

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*⟩

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*⟩

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
⟨*proof*⟩

9.1 Bounding join-work

lemma *diff_size_bound*:
  assumes *inv t1 inv t2*
  shows *size* (*diff t1 t2*) ≤ *size t1*
⟨*proof*⟩

lemma *diff_rank_le_log_size*:
  assumes *inv t1 inv t2*
  shows *rank* (*diff t1 t2*) ≤ c_u ∗ *log 2* (*size1 t1*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *KJk_diff_pos*: *0* < *KJk_diff*
⟨*proof*⟩

lemma *KJc_diff_pos*: *0* < *KJc_diff*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *W_diff_nonneg*: *0* ≤ *W_diff t1 t2*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *joinR_set*:*ht l* > *ht r* + *1* ⟹ *set_tree* (*joinR l a r*) = *set_tree*
l ∪ {*a*} ∪ *set_tree r*
⟨*proof*⟩

lemma *balL_set*:*set_tree* (*balL l a r*) = *set_tree*
l ∪ {*a*} ∪ *set_tree r*
⟨*proof*⟩

lemma *joinL_set*:*ht r* > *ht l* + *1* ⟹ *set_tree* (*joinL l a r*) = *set_tree*
l ∪ {*a*} ∪ *set_tree r*
⟨*proof*⟩

corollary *set_join*:*set_tree* (*join l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *balL_bst*:⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *a*; ∀ *y*∈*set_tree r*. *a* < *y*⟧
⟹ *bst* (*balL l a r*)
⟨*proof*⟩

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*⟩

corollary *bst_join*:*bst* (*Node l* (*a*, *b*) *r*) ⟹ *bst* (*join l a r*)
⟨*proof*⟩

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*⟩

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*⟩

corollary *inv_join*:⟦*avl l*; *avl r*⟧ ⟹ *avl* (*join l a r*)
⟨*proof*⟩

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

interpretation *tree_ht*: *Set2_Join*
where *join* = *join* **and** *inv* = *avl*
⟨*proof*⟩

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*⟩

corollary *rule_mono*: ⟦*avl l*; *avl r*⟧ ⟹ *max* (*height l*) (*height r*) ≤ *height* (*join l a r*)
⟨*proof*⟩

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*⟩

10.3.3 Balancing Rule

interpretation *tree_ht*: *BalancedShape*
where *rank* = *height* **and** c_l = *1* **and** c_u = *2*
⟨*proof*⟩

lemma *rule_bal*: *avl t* ⟹ *tree_ht.balanced t*
⟨*proof*⟩

interpretation *tree_ht*: *BalancedTree*
where *rank* = *height* **and** c_l = *1* **and** c_u = *2* **and** *inv* = *avl*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *T_node_0*[*simp*]: *T_node l a r* = *0*
⟨*proof*⟩

lemma *T_balL_0*[*simp*]: *T_balL l a r* = *0*
⟨*proof*⟩

lemma *T_balR_0*[*simp*]: *T_balR l a r* = *0*
⟨*proof*⟩

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*⟧ ⟹ *T_joinR l a r* ≤ *1* + *height l* − *height r*
⟨*proof*⟩

lemma *T_joinL*:⟦*avl l*; *avl r*; *ht r* > *ht l* + *1*⟧ ⟹ *T_joinL l a r* ≤ *1* + *height r* − *height l*
⟨*proof*⟩

corollary *rule_cost*:
assumes *avl l* ∧ *avl r*
shows *T_join l a r* ≤ *1* + *nat*(*abs*(*int*(*height l*) − *int*(*height r*)))

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *balL_size*:*size* (*balL l a r*) = *size l* + *size r* + *1*
⟨*proof*⟩

lemma *joinR_size*:*ht l* > *ht r* + *1* ⟹ *size* (*joinR l a r*) = *size l* + *size r* + *1*
⟨*proof*⟩

lemma *joinL_size*:*ht r* > *ht l* + *1* ⟹ *size* (*joinL l a r*) = *size l* + *size r* + *1*
⟨*proof*⟩

lemma *join_size*:*size* (*join l a r*) = *size l* + *size r* + *1*
⟨*proof*⟩

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*⟩

10.5 Concrete constants for AVL

All abstract constants are evaluated for the AVL instantiation.

lemma *avl_c_delta*: *tree_ht*.c_δ = *1*
⟨*proof*⟩

lemma *avl_C*: *tree_ht.C* = *1*
⟨*proof*⟩

lemma *avl_K_split*: *tree_ht.K_split* = *8*
⟨*proof*⟩

lemma *avl_K_T*: *tree_ht.K_T* = *4*
⟨*proof*⟩

lemma *avl_K_min*: *tree_ht.K_min* = *4*
⟨*proof*⟩

lemma *avl_K_lin_union*: *tree_ht.K_T* ∗ *tree_ht.K_lin 2 3* ≤ *1238*
⟨*proof*⟩

lemma *avl_K_log_union*: *tree_ht.K_T * tree_ht.K_log 2 3 ≤ 629*
⟨*proof*⟩

lemma *avl_K_lin_inter*: *tree_ht.K_T * tree_ht.K_lin 16 12 ≤ 2222*
⟨*proof*⟩

lemma *avl_K_log_inter*: *tree_ht.K_T * tree_ht.K_log 16 12 ≤ 1131*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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))*
⟨*proof*⟩

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))*
⟨*proof*⟩

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))*
⟨*proof*⟩

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* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
B l a r = *Node l* (*a*, (*Black*, *bh l* + *1*)) *r*

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

⟨*proof*⟩

lemma *neq_Red*[*simp*]: $(c \neq Red) = (c = Black)$
⟨*proof*⟩

11.1.1 Invariants

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

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

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

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

abbreviation *rbt* :: *'a rbt* ⇒ *bool* **where**
rbt t ≡ *invc t* ∧ *invh t*

lemma *invc2I*: *invc t* ⟹ *invc2 t*
⟨*proof*⟩

11.1.2 Rebalancing

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

fun *baliL* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'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* ⇒ *'a* ⇒ *'a rbt* ⇒ *'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*

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set_baliL*: *set_tree* (*baliL l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
⟨*proof*⟩

lemma *set_baliR*: *set_tree* (*baliR l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
⟨*proof*⟩

lemma *bst_baliL*:
⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *a*; ∀ *x*∈*set_tree r*. *a* < *x*⟧ ⟹ *bst* (*baliL l a r*)
⟨*proof*⟩

lemma *bst_baliR*:
⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *a*; ∀ *x*∈*set_tree r*. *a* < *x*⟧ ⟹ *bst* (*baliR l a r*)
⟨*proof*⟩

lemma *size_baliL*: *size* (*baliL l a r*) = *size l* + *size r* + *1*
⟨*proof*⟩

lemma *size_baliR*: *size* (*baliR l a r*) = *size l* + *size r* + *1*
⟨*proof*⟩

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*⟩

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*⟩

A red root is passed upwards unchanged.

lemma *col_joinL*: ⟦*bh l* < *bh r*; *col r* = *Red*⟧ ⟹ *col* (*joinL l x r*) = *Red*
⟨*proof*⟩

lemma *col_joinR*: ⟦*bh r* < *bh l*; *col l* = *Red*⟧ ⟹ *col* (*joinR l x r*) = *Red*
⟨*proof*⟩

lemma *inv_blacken*: ⟦*invc2 t*; *invh t*⟧ ⟹ *invc* (*blacken t*) ∧ *invh* (*blacken t*)
⟨*proof*⟩

lemma *blacken_id*: *invc t* ⟹ *blacken t* = *t*
⟨*proof*⟩

lemma *inv_join*:
assumes *rbt l rbt r*
shows *rbt* (*join l x r*)
⟨*proof*⟩

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*⟩

lemma *set_joinR*: *set_tree* (*joinR l x r*) = *set_tree l* ∪ {*x*} ∪ *set_tree r*
⟨*proof*⟩

lemma *set_blacken*: *set_tree* (*blacken t*) = *set_tree t*
⟨*proof*⟩

lemma *set_join*: *set_tree* (*join l x r*) = *set_tree l* ∪ {*x*} ∪ *set_tree r*
⟨*proof*⟩

lemma *bst_joinL*: *bst* (*Node l* (*a*, *n*) *r*) ⟹ *bst* (*joinL l a r*)
⟨*proof*⟩

lemma *bst_joinR*: *bst* (*Node l* (*a*, *n*) *r*) ⟹ *bst* (*joinR l a r*)
⟨*proof*⟩

lemma *bst_blacken*: *bst* (*blacken t*) = *bst t*
⟨*proof*⟩

lemma *bst_join*: *bst* (*Node l* (*a*, *n*) *r*) ⟹ *bst* (*join l a r*)
⟨*proof*⟩

11.3.3 Size

lemma *size_joinL*: *size* (*joinL l x r*) = *size l* + *size r* + *1*
⟨*proof*⟩

lemma *size_joinR*: *size* (*joinR l x r*) = *size l* + *size r* + *1*
⟨*proof*⟩

lemma *size_blacken*: *size* (*blacken t*) = *size t*
⟨*proof*⟩

lemma *join_size*: *size* (*join l x r*) = *size l* + *size r* + *1*
⟨*proof*⟩

Functional correctness is established by instantiating *Set2_Join*.

interpretation *tree_rb*: *Set2_Join*
where *join* = *join* **and** *inv* = *rbt*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

interpretation *tree_rb*: *BalancedShape*
where *rank* = λ*t*. *real* (*rk t*) **and** c_l = *1* **and** c_u = *2*
⟨*proof*⟩

lemma *rule_bal*: *rbt t* ⟹ *tree_rb.balanced t*
⟨*proof*⟩

interpretation *tree_rb*: *BalancedTree*
where *rank* = λ*t*. *real* (*rk t*) **and** c_l = *1* **and** c_u = *2* **and** *inv* = *rbt*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *T_bh_0*[*simp*]: *T_bh t* = *0*
⟨*proof*⟩

lemma *T_R_0*[*simp*]: *T_R l a r* = *0* **and** *T_B_0*[*simp*]: *T_B l a r* = *0*
⟨*proof*⟩

lemma *T_baliL_0*[*simp*]: *T_baliL l a r* = *0*
⟨*proof*⟩

lemma *T_baliR_0*[*simp*]: *T_baliR l a r* = *0*
⟨*proof*⟩

lemma *T_blacken_0*[*simp*]: *T_blacken t* = *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *rule_cost_EQ*: *bh l = bh r ⟹ T_join l x r = 0*
⟨*proof*⟩

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*⟩

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*⟩

11.6 Concrete constants for red-black trees

lemma *rbt_c_delta*: $tree\_rb.c_\delta = 1$
⟨*proof*⟩

lemma *rbt_C*: *tree_rb.C = 1*
⟨*proof*⟩

lemma *rbt_K_split*: *tree_rb.K_split = 8*
⟨*proof*⟩

lemma *rbt_K_T*: *tree_rb.K_T = 6*
⟨*proof*⟩

lemma *rbt_K_min*: *tree_rb.K_min = 4*
⟨*proof*⟩

lemma *rbt_K_lin_union*: *tree_rb.K_T ∗ tree_rb.K_lin 2 3 ≤ 1856*
⟨*proof*⟩

lemma *rbt_K_log_union*: *tree_rb.K_T ∗ tree_rb.K_log 2 3 ≤ 943*
⟨*proof*⟩

lemma *rbt_K_lin_inter*: *tree_rb.K_T * tree_rb.K_lin 16 12 ≤ 3333*
⟨*proof*⟩

lemma *rbt_K_log_inter*: *tree_rb.K_T * tree_rb.K_log 16 12 ≤ 1696*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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))*
⟨*proof*⟩

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))*
⟨*proof*⟩

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))*
⟨*proof*⟩

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*
⟨*proof*⟩

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⟩
```

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⟩
```

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
⟨proof⟩
```

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
  ⟨proof⟩
```

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.