A Verified Cost Analysis of Joinable Red-Black Trees

Runming Li
runmingl@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Harrison Grodin
hgrodin@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Robert Harper
rwh@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

Ordered sequences of data, specified with a *join* operation to combine sequences, serve as a foundation for the implementation of parallel functional algorithms. This abstract data type can be elegantly and efficiently implemented using balanced binary trees, where a join operation is provided to combine two trees and rebalance as necessary. In this work, we present a verified implementation and cost analysis of joinable red-black trees in **calf**, a dependent type theory for cost analysis. We implement red-black trees and auxiliary intermediate data structures in such a way that all correctness invariants are intrinsically maintained. Then, we describe and verify precise cost bounds on the operations, making use of the red-black tree invariants. Finally, we implement standard algorithms on sequences using the simple join-based signature and bound their cost in the case that red-black trees are used as the underlying implementation. All proofs are formally mechanized using the embedding of **calf** in the Agda theorem prover.

1 Introduction

Ordered sequences of data are essential to the efficient implementation of parallel functional algorithms [Acar and Blelloch 2019]. One common presentation of the signature for ordered sequences containing elements of type α is given in Fig. 1. This signature provides an abstract type seq_α along with three operations:

1. A constructor, EMPTY, that represents the empty sequence containing no data of type α.
2. A constructor, JOIN, that appends two sequences with an element of type α in between.
3. A destructor, REC_ρ, that recurs over a sequence to produce an element of type ρ. An EMPTY sequence is mapped to the argument of type ρ; a sequence $\mathrm{JOIN}(s_1, a, s_2)$ is destructed using the argument of type
$$\mathsf{seq}_\alpha \to \rho \to \alpha \to \mathsf{seq}_\alpha \to \rho \to \rho,$$
plugging s_1 and s_2 in for the sequence arguments, a in for the α argument, and the recursive calls in for the ρ arguments.

These three operations give rise to implementations of all algorithms on ordered sequences of data; some examples are shown in Fig. 2.

Many implementations of this signature are possible, using data structures such as lists and trees. When trees are used, the data in the sequence is taken to be the in-order traversal of the tree. For parallel efficiency, balanced trees are a sensible choice [Blelloch and Greiner 1995]: if the recursor REC_ρ performs both recursive calls in parallel, it is worthwhile to rebalance during a JOIN in preparation for an efficient use of REC_ρ later. As studied by Blelloch et al. [2016, 2022] and Sun [2019], when sequences are implemented as balanced binary trees, implementations of common auxiliary functions on sequences have efficient sequential and parallel cost. For example, sequences may be used as an implementation of finite sets when the stored data is sorted. Then, using EMPTY, JOIN, and REC_ρ, bulk set operations such as union and intersection can be implemented with polylogarithmic span.

type seq_α
$\mathrm{EMPTY} : \mathsf{seq}_\alpha$
$\mathrm{JOIN} : \mathsf{seq}_\alpha \to \alpha \to \mathsf{seq}_\alpha \to \mathsf{seq}_\alpha$
$\mathrm{REC}_\rho : \rho \to (\mathsf{seq}_\alpha \to \rho \to \alpha \to \mathsf{seq}_\alpha \to \rho \to \rho) \to$
$\quad \mathsf{seq}_\alpha \to \rho$

Figure 1. Signature for ordered sequences containing elements of type α.

$\mathrm{SUM} : \mathsf{seq}_{\mathsf{nat}} \to \mathsf{nat}$
$\mathrm{SUM} = \mathrm{REC}_{\mathsf{nat}}\ 0\ (\lambda\ \_\ n_1\ n\ \_\ n_2.\ n_1 + n + n_2)$
$\mathrm{MAP} : (\alpha \to \beta) \to \mathsf{seq}_\alpha \to \mathsf{seq}_\beta$
$\mathrm{MAP}\ f = \mathrm{REC}_{\mathsf{seq}_\beta}\ \mathrm{EMPTY}\ (\lambda\ \_\ s_1\ a\ \_\ s_2.\ \mathrm{JOIN}\ s_1\ (f\ a)\ s_2)$
$\mathrm{REVERSE} : \mathsf{seq}_\alpha \to \mathsf{seq}_\alpha$
$\mathrm{REVERSE} = \mathrm{REC}_{\mathsf{seq}_\alpha}\ \mathrm{EMPTY}\ (\lambda\ \_\ s_1\ a\ \_\ s_2.\ \mathrm{JOIN}\ s_2\ a\ s_1)$

Figure 2. Sample implementations of auxiliary functions on sequences, in terms of EMPTY, JOIN, and REC_ρ.

1.1 Red-black trees

Here, we consider the *red-black tree* (RBT) data structure [Guibas and Sedgewick 1978; Okasaki 1999], a flavor of binary search tree with an elegant functional description and cost analysis. For our purposes, a binary tree is inductive data structure where each inhabitant is either a *leaf* node carrying no data or a *node* carrying a key and two other binary tree children. A red-black tree is a binary tree satisfying the following invariants:

1. every node is colored either red or black;
2. every leaf is considered black;

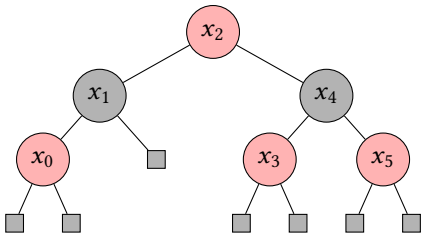

Figure 3. Sample red-black tree with black height of 1. Leaves are depicted as black squares, and nodes are depicted as red or black circles annotated with a key.

3. both children of a red-colored node must be colored black;
4. the number of black nodes on any path from the root to a leaf (excluding the leaf), called the *black height* of the tree, is the same.

Following Blelloch et al. [2016, 2022], we do not require that the root of a red-black tree be colored black. In Fig. 3, we show a sample red-black tree with black height of 1.

Traditionally, red-black trees have been used as binary search trees, storing data in sorted order. Then, the primitive operations are insertion, lookup, and deletion, all of which have similar implementations. However, as discussed by Blelloch et al. [2016, 2022], this causes algorithms implemented using red-black trees to have poor parallel efficiency, since operations must be performed one-at-a-time. Instead, *op. cit.*, a JOIN operation for red-black trees is given, combining two trees with a middle key and rebalancing as necessary to meet the red-black invariants and preserve the in-order traversal ordering. In Fig. 4, we show two sample red-black trees t_1 and t_2 which, when joined with x_5 in the middle, produce the tree t.

It is well-known that red-black trees intrinsically satisfying the above invariants can be defined inductively [Licata 2013; Wang et al. 2017; Weirich 2014]:

1. A black-colored RBT with black height 0, a leaf, may always be formed.
2. Let t_1 and t_2 are black-colored RBTs with black height n, and let a be a key. Then, a red-colored RBT with black height n may be formed.
3. Let t_1 and t_2 be RBTs with black height n, and let a be a key. Then, a black-colored RBT with black height $n+1$ may be formed.

We will use this presentation of red-black trees in our definitions and analysis.

1.2 Mechanized cost analysis in calf

The cost-aware logical framework (**calf**) [Niu et al. 2022] is a dependent type theory for verifying the sequential and parallel cost and correctness of algorithms. **calf** is based on the call-by-push-value paradigm [Levy 2003], separating

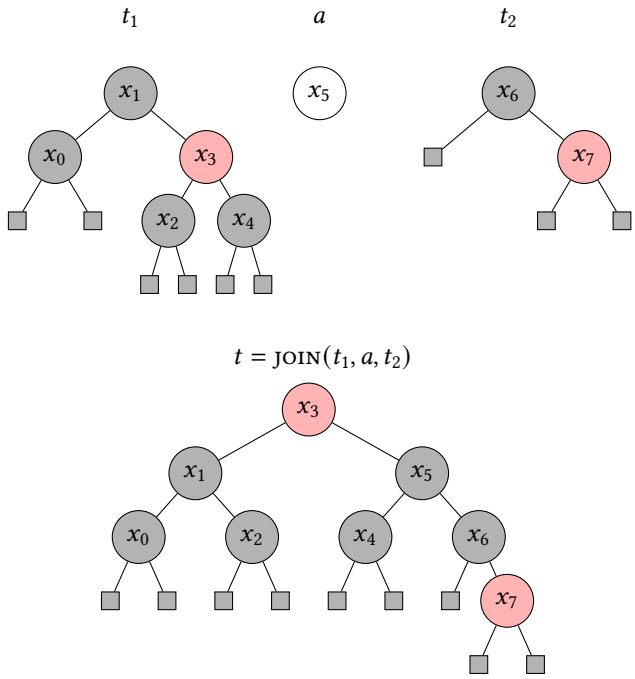

Figure 4. Two red-black trees t_1 and t_2 along with the tree t produced when they are joined with $a = x_5$ in the middle.

$$
\begin{aligned}
&\text{seq}_\alpha : \text{tp}^+ \\
&\textsc{empty} : \mathsf{U}(\text{seq}_\alpha) \\
&\textsc{join} : \mathsf{U}(\text{seq}_\alpha \to \alpha \to \text{seq}_\alpha \to \mathsf{F}(\text{seq}_\alpha)) \\
&\textsc{rec}_\rho : \mathsf{U}(\\
&\quad \mathsf{U}(\rho) \\
&\quad \to \mathsf{U}(\text{seq}_\alpha \to \mathsf{U}(\rho) \to \alpha \to \text{seq}_\alpha \to \mathsf{U}(\rho) \to \rho) \\
&\quad \to \text{seq}_\alpha \to \rho \\
&)
\end{aligned}
$$

Figure 5. Signature for ordered sequences containing elements of value type $\alpha : \text{tp}^+$, for computation types $\rho : \text{tp}^\ominus$.

computations (which may have an associated cost) from values. Computation types are elements of the universe $\text{tp}^\ominus$, whereas value types are elements of the universe tp^+. Function types are computation types, where the input type is a value type and the output type is a computation type. In this setting, the signature for ordered sequences from Fig. 1 is augmented to include $\mathsf{U}(-)$ and $\mathsf{F}(-)$ type constructors, explicitly moving between value and computation types; this change is rendered in Fig. 5.

In **calf**, the programmer includes cost annotations within algorithms, denoting an abstract notion of cost to later analyze. In this work, we use the usual sequential-and-parallel cost model [Niu et al. 2022, §6], where a cost is a pair of the sequential work and the parallel span as natural numbers. To annotate a program with c (sequential and parallel) cost, we write **step** c.

Originally, Niu et al. [2022] studied the implementation of sequential and parallel algorithms on concrete data structures in **calf**. In subsequent work, Grodin and Harper [2023] consider the analysis of sequential-use data structures in this setting. Here, we begin to investigate the implementation and analysis of parallel data structures in **calf**.

1.3 Contribution

In this work, we present an implementation of sequences using joinable red-black trees in **calf**. The correctness of our implementation is intrinsically verified, and we perform a separate precise cost analysis in terms of the number of recursive calls. Following Blelloch et al. [2016, 2022], we implement a variety of sequence functions generically in the given primitives, and we analyze the cost of a simple function in the case the underlying implementation of the sequence type is the red-black tree data structure.

Our implementation and proofs are fully mechanized in Agda [Norell 2009], in which **calf** is embedded [Niu et al. 2022]. We implement the mechanization of sequences and red-black trees in `Examples/Sequence.agda` and the corresponding `Examples/Sequence` directory.

1.4 Related work

Join-based balanced binary trees have been studied extensively by Blelloch et al. [2016, 2022], and the joinable framework is unified by Sun [2019].

The correctness of red-black trees with their traditional sequential operations, such as single-element insertion, have been intrinsically (and extrinsically) verified in a variety of verification environments, including Agda [Licata 2013; Weirich 2014], Coq [Appel 2011, 2023], and Isabelle [Nipkow 2023]. However, these systems do not come equipped with a notion of cost, preventing the verification of the efficiency of these algorithms:

> Coq does not have a formal time–cost model for its execution, so we cannot verify [the] logarithmic running time [of insertion and lookup on red-black trees] in Coq. [Appel 2023]

In another direction, the cost analysis of sequential operations on red-black trees has been verified in a resource-aware type theory [Wang et al. 2017]. However, this work does not verify the correctness of the data structure.

In this work, we verify both the correctness and cost of joinable red-black trees using an abstract cost model in **calf**; further explanation of and examples in the **calf** framework are presented in the original work of Niu et al. [2022].

2 Intrinsically-correct definitions

In this section, we describe a binary tree data type that structurally guarantees that the red-black invariants hold. Then, we describe how it would be used to implement the sequence signature of Fig. 5; of particular interest is the implementation of the JOIN algorithm. Since our definitions will be well-typed, they will be intrinsically correct. We work in **calf**, an extension of call-by-push-value, in which we distinguish value types in universe tp^+ from computation types in universe $\mathsf{tp}^\ominus$.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{irbt}_\alpha : \mathsf{color} \to \mathsf{nat} \to \mathsf{list}(\alpha) \to \mathsf{tp}^+\ \textbf{where}\\
&\quad \mathsf{leaf} : \mathsf{irbt}_\alpha\ \mathsf{black}\ \mathsf{zero}\ []\\
&\quad \mathsf{red} : (\mathsf{irbt}_\alpha\ \mathsf{black}\ n\ l_1)\ (a : \alpha)\ (\mathsf{irbt}_\alpha\ \mathsf{black}\ n\ l_2)\\
&\qquad \to \mathsf{irbt}_\alpha\ \mathsf{red}\ n\ (l_1 +\!\!+ [a] +\!\!+ l_2)\\
&\quad \mathsf{black} : (\mathsf{irbt}_\alpha\ y_1\ n\ l_1)\ (a : \alpha)\ (\mathsf{irbt}_\alpha\ y_2\ n\ l_2)\\
&\qquad \to \mathsf{irbt}_\alpha\ \mathsf{black}\ \mathsf{suc}(n)\ (l_1 +\!\!+ [a] +\!\!+ l_2)\\
&\\
&\mathsf{rbt}_\alpha : \mathsf{tp}^+\\
&\mathsf{rbt}_\alpha = \textstyle\sum_{y:\mathsf{color}} \sum_{n:\mathsf{nat}} \sum_{l:\mathsf{list}(\alpha)} \mathsf{irbt}_\alpha\ y\ n\ l
\end{aligned}
$$

Figure 6. Definition of indexed red-black trees as an indexed inductive type.

First, we define red-black trees as an indexed inductive type, as described in Section 1.1, guaranteeing that the red-black invariants are maintained; this definition of irbt_α is given in Fig. 6. We include an index storing the in-order traversal of the tree that we will to use to guarantee that well-typed definitions implement the desired behavior, specified in terms of lists. Additionally, we define the type rbt_α as the total space of the type family irbt_α, storing an arbitrary color, black-height, and in-order traversal along with an indexed red-black tree with those parameters.

Given these definitions, the goal is to implement the sequence signature of Fig. 5. We choose $\mathsf{seq}_\alpha = \mathsf{rbt}_\alpha$, define $\textsc{empty} = \mathsf{ret}(\mathsf{leaf})$, and naturally implement $\textsc{rec}_\rho$ via the induction principle for rbt_α. It remains, then, to define a computation

$$\textsc{join} : \mathsf{rbt}_\alpha \to \alpha \to \mathsf{rbt}_\alpha \to \mathsf{F}(\mathsf{rbt}_\alpha),$$

which we consider in the remainder of this section.

2.1 The JOIN algorithm

The algorithm itself will follow Blelloch et al. [2016], although we must ensure that the intrinsic structural properties are valid.[1] We recall its definition in Algorithm 1, adapting to our notation; it is defined in terms of auxiliary functions JOINRIGHT (and the symmetric JOINLEFT, which we henceforth elide), which we will consider in the next section. Informally, the algorithm proceeds as follows:

1. If both trees have equal height, simply construct a new node without rebalancing (Fig. 7). If possible, a red node is preferable.

[1] We omit a case listed by Blelloch et al. [2022] that our verification shows is impossible to reach.

Algorithm 1 JOIN algorithm for red-black trees [Blelloch et al. 2022]. Since the color and black-height outputs may be inferred, we leave them implicit for readability.

Input:
$t_1 : \mathsf{irbt}_\alpha\ y_1\ n_1\ l_1$
$a : \alpha$
$t_2 : \mathsf{irbt}_\alpha\ y_2\ n_2\ l_2$
Output:
$\textsc{Join}(t_1, a, t_2) : \mathsf{F}(\sum_{y:\mathsf{color}} \sum_{n:\mathsf{nat}} \mathsf{irbt}_\alpha\ y\ n\ (l_1 +\!\!+ [a] +\!\!+ l_2))$

switch $\textsc{Compare}(n_1, n_2)$ **do**
case $n_1 > n_2$
  $t' \leftarrow \textsc{JoinRight}(t_1, a, t_2)$
  switch t' **do**
  case meets the invariants
    return t'
  case has a red-red violation on the right
    red $(t'_1,\ a',\ t'_2) \leftarrow t'$
    return black $(t'_1,\ a',\ t'_2)$
case $n_1 < n_2$
  $\cdots$ ▷ symmetric, in terms of JOINLEFT
case $n_1 = n_2$
  if $y_1 =$ black and $y_2 =$ black **then**
    return red $(t_1,\ a,\ t_2)$
  else
    return black $(t_1,\ a,\ t_2)$
  end if

2. Otherwise, without loss of generality, assume t_1 has a larger black height than t_2. Then, use the JOINRIGHT auxiliary function to place t_2 on the right spine of t_1, rebalancing as necessary. The process may cause a single red-red violation at the root of the result tree. In that case, recolor the root to black (Fig. 8); otherwise, return the valid tree.

This algorithm performs no recursive calls aside from those within JOINRIGHT, so no cost annotations are required by our cost model. It remains, then, to define the type and implementation of JOINRIGHT.

2.2 The JOINRIGHT auxiliary algorithm

As discussed previously, the JOINRIGHT algorithm has a relaxed specification: rather than guaranteeing a valid red-black tree, it allows a single red-red violation between the root of the result and its right child to propagate upwards. We allow this violation only in the case that the first tree had a red root to begin with.

In order to represent this condition, we define an auxiliary data structure, an *almost-right red-black tree*, abbreviated arrbt, in Fig. 9; our terminology is inspired by the "almost tree" of Weirich [2014]. A well-formed red-black tree always counts as an almost-right red-black tree; a

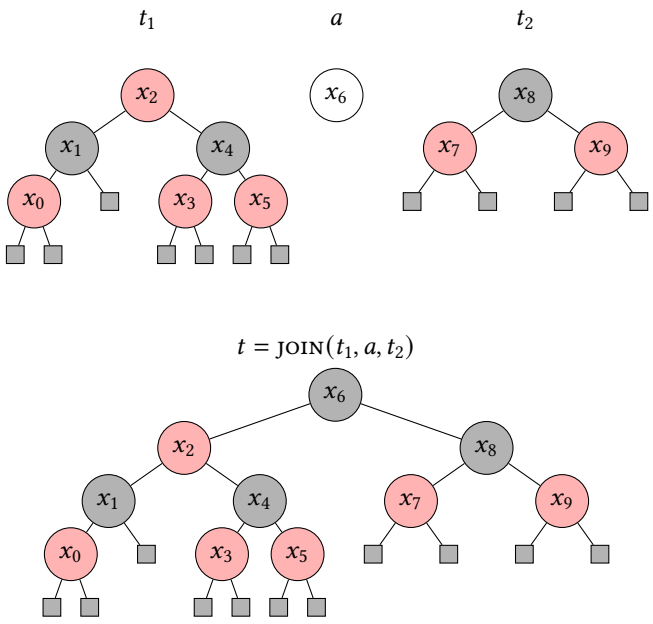

Figure 7. Join of two trees with equal black heights.

red-colored almost-right red-black tree may also be a violation, with a black-colored left child, key data, and another red-colored right child. Notably, a red-red violation for an almost-right red-black tree can only happen on the right spine, and only when the first tree originally had a red root. We thereby define arrbt to be indexed by another color parameter called leftColor, representing the color of the left tree from which it was created. Therefore, when a violation happens, the leftColor must be red. Given this definition, we wish to define a computation

$\textsc{JoinRight}$:
$(\mathsf{irbt}_\alpha\ y_1\ n_1\ l_1)\ (a : \alpha)\ (\mathsf{irbt}_\alpha\ y_2\ n_2\ l_2) \to$
$n_1 > n_2 \to$
$\mathsf{F}(\mathsf{arrbt}_\alpha\ y_1\ n_1\ (l_1 +\!\!+ [a] +\!\!+ l_2))$.

Observe that the black height and left color of the result must match the first tree. Also, notice that given such a definition of JOINRIGHT, the JOIN implementation of Algorithm 1 is well-typed and therefore correct.

Lemma 2.1. *For all well-typed t_1, a, and t_2, it is the case that $\textsc{Join}(t_1, a, t_2)$ is a valid red-black tree.*

Proof. We assume a well-typed implementation of JOINRIGHT, which is provided in Algorithm 2 and proved correct in Lemma 2.2.

If $n_1 > n_2$, the call $\textsc{JoinRight}(t_1, a, t_2)$ is made, returning an almost-right red-black tree. If this tree is valid, this tree is returned, as desired. Otherwise, if it has a red-red violation between the root and its right child, then the root is changed to black, causing all the red-black invariants to be satisfied.

If $n_1 < n_2$, then a symmetric argument can be made.

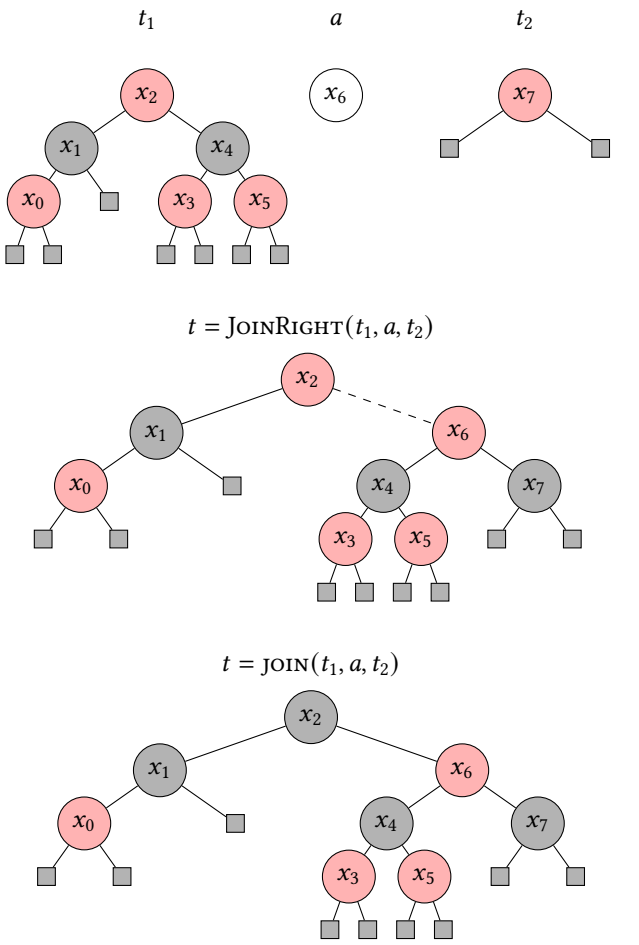

Figure 8. Recoloring the root of a result tree from JoinRight due to a red-red violation on the right, indicated by a dashed line.

```
data arrbt_α : color → nat → list(α) → tp⁺ where
  valid : (leftColor : color) (irbt_α y n l)
    → arrbt_α leftColor n l
  violation : (irbt_α black n l₁) (a : α) (irbt_α red n l₂)
    → arrbt_α red n (l₁ ++ [a] ++ l₂)
```

Figure 9. Definition of almost-right red-black trees, allowing for a red-red violation on the right when the color parameter (the color of the left tree from which it was created) is red, as an indexed inductive type.

If $n_1 = n_2$, then the two trees may be joined by a red node if both are black or a black node otherwise. In either case, it forms a valid red-black tree. □

Now, it remains to give the JoinRight algorithm to fulfill this specification. Here, we diverge slightly from Blelloch et al. [2016, 2022] for ease of verification. The algorithm presented *op. cit.* allows for a triple-red violation on the right spine, albeit only in the base case. Moreover, as noted by Sun [2019, §3.2.2], the triple-red issue must be resolved one recursive call after the base case. Therefore, we trade the more concise code and more complex specification for slightly more verbose code with a simpler specification. We give our definition of JoinRight in Algorithm 2.

We claim that JoinRight is a well-typed program with exhaustive casework, by the definitions of irbt_α and arrbt_α. Although our Agda mechanization verifies this fact, we include an informal proof below.

Lemma 2.2. *For all appropriate inputs t_1, a, and t_2, the call $\textsc{JoinRight}(t_1, a, t_2)$ returns an almost-right red-black tree with black height n_1. In other words:*

1. *If t_1 is colored* black*, then $\textsc{JoinRight}(t_1, a, t_2)$ is a valid red-black tree with the same black height as t_1.*
2. *If t_1 is colored* red*, then $\textsc{JoinRight}(t_1, a, t_2)$ is an almost-right red-black tree (valid or with a red-red violation) with the same black height as t_1.*

Proof. We prove both items simultaneously by induction on t_1, following the structure of the code.

I. If t_1 is colored red, we must prove Item 2, and its children $t_{1,1}$ and $t_{1,2}$ must both be colored black. Moreover, $n_1 = n_{1,1} = n_{1,2} > n_2$. By induction, the result of the recursive call to $\textsc{JoinRight}(t_{12}, a, t_2)$, t', gives a valid red-black tree with black height $n_{1,2}$. We always return a red node whose left child is the black subtree $t_{1,1}$ and whose right child is t', which could be either red or black. Depending on the color of t', we will either get a valid red tree or a red-red violation on the right spine, both of which are allowed as the result for Item 2.

II. If t_1 is colored black, we must prove Item 1. If $n_1 = n_2+1$ and t_2 is colored red, then $n_1 = n_{2,1}+1 = n_{2,2}+1$. Therefore, the returned tree is valid with black height n_1.

III. This case is similar to the previous case, but t_2 is colored black. If $t_{1,2}$ is colored red, then $n_{1,1} = n_{1,2} = n_{2,1} = n_{2,2} = n_2$. Therefore, the returned tree is valid with black height n_1.

IV. This case is similar to the previous case, but $t_{1,2}$ is colored black. Thus, $n_{1,1} = n_{1,2} = n_{1,2,1} = n_{1,2,2} = n_2$, so the returned tree is valid with black height n_1.

V. If t_1 is colored black, we must prove Item 1. Suppose $n_1 > n_2 + 1$. Then, $n_{1,1} + 1 = n_{1,2} + 1 = n_1 > n_2$. Regardless of the color of $t_{1,2}$, the inductive hypothesis applies. If the result r is a valid red-black tree t', then $t_{1,1}$ and a_1 can be combined at a black node to create a valid red-black tree with black height n_1.

VI. This case is similar to the previous case, but the result r indicates a red-red violation between the root and its right child. Then, a left-rotation is performed to give

Algorithm 2 JoinRight algorithm for red-black trees, based on Blelloch et al. [2022]. Cases are exhaustive, by the definitions of irbt_α and arrbt_α, with the outer induction on t_1. Cost annotations are highlighted.

Input:
$t_1 : \mathsf{irbt}_\alpha\ y_1\ n_1\ l_1$
$a : \alpha$
$t_2 : \mathsf{irbt}_\alpha\ y_2\ n_2\ l_2$
$n_1 > n_2$
Output:
$\textsc{JoinRight}(t_1, a, t_2) : \mathsf{F}(\mathsf{arrbt}_\alpha\ y_1\ n_1\ (l_1 +\!\!+ [a] +\!\!+ l_2))$

switch t_1 **do**
case $t_1 =$ red $(t_{1,1},\ a_1,\ t_{1,2})$ ▷ Case I.
  step 1
  $\mathsf{valid}(t') \leftarrow \textsc{JoinRight}(t_{1,2}, a, t_2)$
  switch y', the color of t' **do**
  case $y' =$ red
    return $\mathsf{violation}(t_{1,1},\ a_1,\ t')$
  case $y' =$ black
    return $\mathsf{valid}($ red $(t_{1,1},\ a_1,\ t'))$
case $t_1 =$ black $(t_{1,1},\ a_1,\ t_{1,2})$
  switch compare n_1 and n_2 **do**
  case $n_1 = n_2 + 1$
    switch t_2 **do**
    case $t_2 =$ red $(t_{2,1},\ a_2,\ t_{2,2})$ ▷ Case II.
      return $\mathsf{valid}($ red $(t_1,\ a,$ black $(t_{2,1},\ a_2,\ t_{2,2})))$
    case $t_2 =$ black $(t_{2,1},\ a_2,\ t_{2,2})$
      switch $t_{1,2}$ **do**
      case $t_{1,2} =$ red $(t_{1,2,1},\ a_{1,2},\ t_{1,2,2})$ ▷ Case III.
        $x_1 \leftarrow$ black $(t_{1,1},\ a_1,\ t_{1,2,1})$
        $x_2 \leftarrow$ black $(t_{1,2,2},\ a,\ t_2)$
        return $\mathsf{valid}($ red $(x_1,\ a_{1,2},\ x_2))$
      case $t_{1,2} =$ black $(t_{1,2,1},\ a_{1,2},\ t_{1,2,2})$ ▷ Case IV.
        $x_2 \leftarrow$ red $(t_{1,2},\ a,\ t_2)$
        return $\mathsf{valid}($ black $(t_{1,1},\ a_1,\ x_2))$
  case $n_1 > n_2 + 1$
    step 1
    $r \leftarrow \textsc{JoinRight}(t_{1,2}, a, t_2)$
    switch r **do**
    case $r = \mathsf{valid}(t')$
      return $\mathsf{valid}($ black $(t_{1,1},\ a_1,\ t'))$ ▷ Case V.
    case $r = \mathsf{violation}(t'_1,\ a',$ red $(t'_{2,1},\ a'_2,\ t'_{2,2}))$
      $x_1 \leftarrow$ black $(t_{1,1},\ a_1,\ t'_1)$
      $x_2 \leftarrow$ black $(t'_{2,1},\ a'_2,\ t'_{2,2})$
      return $\mathsf{valid}($ red $(x_1,\ a',\ x_2))$ ▷ Case VI.

back a valid red-colored red-black tree with black height n_1.

In every case, the in-order traversal of the tree is clearly preserved, by inspection of the left-to-right order of the subtrees and keys. □

Thus, we have described the Join algorithm on red-black trees and intrinsically verified its correctness. Based on the correctness of JoinRight, we also get a straightforward bound on the black height of the tree produced by Join, matching the result of Blelloch et al. [2016, 2022].

Theorem 2.3. *Let t_1 and t_2 be red-black trees with black heights n_1 and n_2, respectively. Then, the black height of the red-black tree returned by $\textsc{Join}(t_1, a, t_2)$ is either $\max(n_1, n_2)$ or $1 + \max(n_1, n_2)$.*

Theorem 2.3 does not affect the cost analysis of Join, but it does impact cost analysis for algorithms that use Join; therefore, it is also mechanized in the implementation.

For the purpose of correctness analysis, the cost annotations did not play a role. In the next section, we will state and prove cost bounds on the Join and JoinRight algorithms.

3 Cost analysis

To analyze the cost of algorithms in **calf**, we attempt to bound the number of calls to **step**. In the subsequent development, we will count informally; in our mechanization, we use the definition isBounded $(A; e; c)$ and associated lemmas from the **calf** standard library [Niu et al. 2022]. From this section onward, we annotate all mechanized results with their name as defined in the Agda implementation using the typewriter font, *e.g.* `joinRight/is-bounded`.

3.1 Cost of JoinRight

If a red-black tree has black height n, it has true height bounded by at most $2n + 1$: on top of every black node, an additional red node may (optionally) be placed without affecting the black height. Similar, then, to how an almost-right red-black tree weakens the invariants in the case of a red root, so too must the cost analysis weaken the cost bound given a red root.

Theorem 3.1 (`joinRight/is-bounded`)**.** *Let t_1, a, and t_2 be valid inputs to JoinRight. Then, the cost of $\textsc{JoinRight}(t_1, a, t_2)$ is bounded by $1 + 2(n_1 - n_2)$.*

Proof. We prove a strengthened claim:

1. If t_1 is colored red, the cost of $\textsc{JoinRight}(t_1, a, t_2)$ is bounded by $1 + 2(n_1 - n_2)$.
2. If t_1 is colored black, the cost of $\textsc{JoinRight}(t_1, a, t_2)$ is bounded by $2(n_1 - n_2)$.

The desired result follows immediately in both cases. Following the structure of the JoinRight in Algorithm 2, we go by induction on t_1.

I. Since t_1 is colored red, $t_{1,2}$ is black with $n_1 = n_{1,2}$, and we must prove Item 1. This case incurs 1 cost in

addition to the cost of the recursive call. The cost of the recursive call is bounded by $2(n_{1,2} - n_2) = 2(n_1 - n_2)$. Therefore, the cost of the entire computation is bounded by $1 + 2(n_1 - n_2)$, as desired.

II. This case incurs zero cost.

III. This case incurs zero cost.

IV. This case incurs zero cost.

V. Since t_1 is colored black, $n_1 = n_{1,2} + 1$, and we must prove Item 2. This case incurs 1 cost in addition to the cost of the recursive call. The color of $t_{1,2}$ is unknown, but in either case the cost of the recursive call is bounded by $1+2(n_{1,2}-n_2)$. Therefore, the cost of the entire computation is bounded by $2+2(n_{1,2}-n_2) = 2((n_{1,2}+1)-n_2) = 2(n_1-n_2)$, as desired.

VI. This case is the same as the previous case.

In all cases, the desired result holds. □

3.2 Cost of JOIN

Using Theorem 3.1, we may now reason about the cost of the full JOIN implementation of Algorithm 1. For notational convenience, we write

$$\overline{x_1} = \max(x_1, x_2)$$
$$\overline{x_2} = \min(x_1, x_2)$$

since JOIN behaves symmetrically depending on which tree is larger.

Theorem 3.2 (join/is-bounded). *For all t_1, a, and t_2, the cost of $\mathit{JOIN}(t_1, a, t_2)$ is bounded by $1 + 2(\overline{n_1} - \overline{n_2})$.*

Proof. If t_1 and t_2 have the same black height, no cost is incurred, so the bound is trivially met. Otherwise, the result follows immediately from Theorem 3.1. □

This validates the claim by Blelloch et al. [2022, §4.2] that the cost of JOIN on red-black tree is in $O(|h(t_1) - h(t_2)|)$, where $h(t)$ is the height of tree t.

Since black height is a property only understood in the implementation, rather than the abstract sequence interface, we wish to publicly characterize the cost of JOIN in terms of the lengths of the involved sequences. To accomplish this, we bound the black height of a red-black tree in terms of the overall size of the tree, which we write $|t|$ for a tree t.

Lemma 3.3 (nodes/upper-bound). *For any red-black tree t with black height n, we have*

$$n \le \lceil \log_2(1 + |t|) \rceil.$$

Lemma 3.4 (nodes/lower-bound). *For any red-black tree t with black height n, we have*

$$\left\lfloor \frac{\lceil \log_2(1 + |t|) \rceil - 1}{2} \right\rfloor \le n.$$

Using these lemmas, we may give a user-facing description of the cost of JOIN.

Algorithm 3 Recursive SUM algorithm for sequences. Pattern-matching syntax for EMPTY and JOIN is syntactic sugar for $\mathrm{REC}_{\mathsf{F(nat)}}$.

Input:
$s : \mathsf{seq}_{\mathsf{nat}}$
Output:
$\textsc{Sum}(s) : \mathsf{F(nat)}$

switch s **do**
case EMPTY
  return 0
case $\textsc{Join}(s_1, a, s_2)$
  step 1
  $(x_1, x_2) \leftarrow \textsc{Sum}(s_1) \parallel \textsc{Sum}(s_2)$
  return $x_1 + a + x_2$

Theorem 3.5 (join/is-bounded/nodes). *Let t_1, a, and t_2 be valid inputs to JOIN. Then, the cost of $\mathit{JOIN}(t_1, a, t_2)$ is bounded by*

$$1 + 2\left(\left\lceil \log_2(1 + \overline{|t_1|}) \right\rceil - \left\lfloor \frac{\left\lceil \log_2(1 + \overline{|t_2|}) \right\rceil - 1}{2} \right\rfloor\right).$$

This matches the expected cost bound,

$$O\left(\left\lceil \log_2\left(\overline{|t_1|}/\overline{|t_2|}\right)\right\rceil\right).$$

4 Case study: algorithms on sequences

An essential part of the work of Blelloch et al. [2016, 2022] and Sun [2019] is showing how an implementation of the sequence signature gives rise to efficient implementations of other common algorithms on sequences when sequences are implemented as balanced trees. Here, we consider the implementation and cost analysis of some such algorithms. We implement each algorithm generically in terms of the sequence interface given in Fig. 5. However, for the purpose of cost analysis, we break abstraction, inlining the sequence definitions. Additionally, for readability, we replace uses of REC_ρ with a more familiar pattern matching notation.

4.1 Sequence sum

One simple algorithm on a sequence of natural numbers is a parallel sum, adding up the elements in linear work and logarithmic span with respect to the length of the sequence when counting recursive calls. We give an implementation

$$\textsc{Sum} : \mathsf{seq}_{\mathsf{nat}} \to \mathsf{F(nat)}$$

in Algorithm 3, adapting the definition from Fig. 2 to the call-by-push-value setting and adding cost instrumentation and parallelism. It goes by recursion using $\mathrm{REC}_{\mathsf{F(nat)}}$. In the base case, 0 is returned. In the inductive case, it recursively sums both subsequences *in parallel* and then returns the sum of the results and the middle datum.

When the implementation of sequences is specialized to red-black trees, we achieve the desired cost bound.

Theorem 4.1 (sum/bounded). *For all red-black trees t, the cost of $\textsc{Sum}(t)$ is bounded by*

- $|t|$ *work (sequential cost) and*
- $1 + 2\lceil \log_2(1 + |t|) \rceil$ *span (idealized parallel cost).*

Proof. The sequential bound is immediate by induction. The parallel bound is shown using the black height, showing a bound of $1 + 2n$ (and a strengthened bound of $2n$ in case the tree is black) by induction. Then, Lemma 3.3 translates the bound from black height to the size of the tree. □

This matches the result of Blelloch et al. [2016, 2022]: linear work and logarithmic span.

4.2 Finite set functions

Blelloch et al. [2016, 2022] consider implementations of standard functions on finite sets using balanced trees. Here, we briefly show how such implementations could be provided in terms of the basic sequence signature of Fig. 5.

In order to implement a finite set as a sequence, we assume the element type α is equipped with a total order. Then, standard functions on finite sets may be implemented using the recursor on sequences. In Fig. 10, we provide generic implementations of some examples:

1. The Split function splits a sorted sequence at a designated value, providing the elements of the sequence less than and greater than the value and, if it exists, the equivalent value.
2. The Insert function inserts a new value into the correct position in a sorted sequence, simply splitting the sequence at the desired value and joining the two sides around the new value.
3. The Union function takes the union of two sorted sequences, combining their elements to make a new sorted sequence.

Blelloch et al. study the efficiency of these and other similar algorithms is studied, showing that implementations in terms of empty, join, and $\textsc{rec}_\rho$ have comparable efficiency to bespoke definitions. We include the implementations of these algorithms in our mechanization, but we leave their cost and correctness verification to future work.

```
Split : seq_α → α → F(seq_α × option(α) × seq_α)
Split s a =
  REC_{F(seq_α × option(α) × seq_α)}
    ret(EMPTY, none, EMPTY)
    (λ s1 r1 a' s2 r2.
      compare a a' of
        = : ret(s1, some(a), s2)
        < : (s1,1, a?, s1,2) ← r1;
            s' ← JOIN(s1,2, a', s2);
            ret(s1,1, a?, s')
        > : (s2,1, a?, s2,2) ← r2;
            s' ← JOIN(s1, a', s2,1);
            ret(s', a?, s2,2))
    s
Insert : seq_α → α → F(seq_α)
Insert s a = (s1, a?, s2) ← Split s a; JOIN(s1, a, s2)
Union : seq_α → seq_α → F(seq_α)
Union =
  REC_{seq_α → F(seq_α)}
    (λs. s)
    (λ _ f1 a _ f2. λs2.
      (s2,1, a?, s2,2) ← Split s2 a;
      (u1, u2) ← f1 s2,1 ‖ f2 s2,2;
      JOIN(u1, a', u2))
```

Figure 10. Sample implementations of functions on sequences that use empty, join, and $\textsc{rec}_\rho$.

5 Conclusion

In the work, we presented an implementation of the join algorithm on red-black trees Blelloch et al. [2016, 2022] whose correctness is intrinsically verified due to structural invariants within the type definition. Our implementation was given in **calf**, instrumented with cost annotations to count the number of recursive calls performed; using the techniques developed by Niu et al. [2022], we gave a formally verified precise cost bound proof for the join algorithm.

As noted by Blelloch et al. [2016, 2022], balanced trees are an appealing choice for the implementation of persistent sequences. Since the join-based presentation of sequences provides an induction principle over the underlying balanced trees, where call-by-push-value suspends the results of recursive calls, we were able to implement standard functional algorithms on sequences and, following Blelloch et al., prove their efficient sequential and parallel cost bounds.

5.1 Future work

In this work, we begin to study parallel-ready data structures. This suggests a myriad of directions for future work.

Full sequence library. A natural next step following from this work would be the verification of correctness conditions and cost bounds on other algorithms included in persistent sequence libraries.

Finite sets and dictionaries. Another common use case of balanced trees, as explored in depth by Blelloch et al. [2016, 2022], is the implementation of finite sets and dictionaries by imposing and maintaining a total order on the data stored in the tree. In Section 4.2, we briefly discuss the implementation of finite sets using sorted sequences; as future work, we hope to extend this development to a full-scale finite set library with cost and correctness verification.

Amortized complexity. Although we study the binary JOIN operation on red-black trees in this work, more common historically is the single-element insertion operation. Once the desired location for the new element is found, insertion into the tree along with any necessary rebalancing has asymptotically constant amortized cost [Tarjan 1983]. We expect this result could be verified similarly to other amortized analyses in **calf** [Grodin and Harper 2023].

Various balancing schemes. Blelloch et al. [2016, 2022] study a variety of tree balancing schemes, including AVL trees, weight-balanced trees, and treaps. All of these balancing schemes match the sequence signature, as well; we hope to implement and verify these schemes in future work. Unlike red-black trees, some of these schemes cannot be implemented purely functionally, *e.g.* treaps. This suggests an extension of **calf** that can better take effects into account.

Modular analysis of large-scale algorithms. Many functional algorithms are implemented based on sequences, finite sets, and dictionaries [Acar and Blelloch 2019]. However, in this work, we were forced to reveal the implementation of sequences as red-black trees in order to analyze the efficiency of algorithms implemented generically, such as SUM. In general, such analyses may even depend on particular hidden invariants within an implementation type; thus, we anticipate that analysis of larger-scale algorithms in this fashion would be intractable. Going forward, we hope to further develop a theory of modularity for algorithm cost, allowing algorithms implemented in terms of abstract data types to be analyzed without fully revealing the implementation of the abstraction.

Acknowledgments

We are grateful to Guy Blelloch for insightful discussions and advice about this work.

This work was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-21-0009 (Tristan Nguyen, program manager) and the National Science Foundation under award number CCF-1901381. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR or the NSF.

References

Umut A. Acar and Guy E. Blelloch. 2019. *Algorithms: Parallel and Sequential.* http://www.algorithms-book.com.

Andrew W. Appel. 2011. Efficient Verified Red-Black Trees. (2011).

Andrew W. Appel. 2023. *Verified Functional Algorithms.* Software Foundations, Vol. 3. Electronic textbook. http://softwarefoundations.cis.upenn.edu Version 1.5.4.

Guy Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95).* Association for Computing Machinery, New York, NY, USA, 226–237. https://doi.org/10.1145/224164.224210

Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (Pacific Grove, California, USA) *(SPAA '16).* Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2935764.2935768

Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Trans. Parallel Comput.* 9, 2, Article 7 (apr 2022), 41 pages. https://doi.org/10.1145/3512769

Harrison Grodin and Robert Harper. 2023. Amortized Analysis via Coinduction. In *10th Conference on Algebra and Coalgebra in Computer Science (CALCO 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 270)*, Paolo Baldan and Valeria de Paiva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:6. https://doi.org/10.4230/LIPIcs.CALCO.2023.23

Leo J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science (Sfcs 1978).* 8–21. https://doi.org/10.1109/SFCS.1978.3

Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis.* Springer Netherlands, Dordrecht. https://doi.org/10.1007/978-94-007-0954-6

Dan Licata. 2013. Programming and Proving in Agda, OPLSS 2013. http://www.cs.cmu.edu/~drl/teaching/oplss13/

Tobias Nipkow (Ed.). 2023. *Functional Data Structures and Algorithms: A Proof Assistant Approach* (june 27, 2023 ed.). https://functional-algorithms-verified.org/

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A Cost-Aware Logical Framework. *Proc. ACM Program. Lang.* 6, POPL, Article 9 (jan 2022), 31 pages. https://doi.org/10.1145/3498670

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09).* Savannah, GA, USA, 1–2.

Chris Okasaki. 1999. Red-Black Trees in a Functional Setting. *Journal of Functional Programming* 9, 4 (July 1999), 471–477. https://doi.org/10.1017/S0956796899003494

Yihan Sun. 2019. *Join-based Parallel Balanced Binary Trees.* Ph.D. Dissertation. Carngie Mellon University.

Robert Endre Tarjan. 1983. 4. Search Trees. In *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 45–57. https://doi.org/10.1137/1.9781611970265.ch4

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 79:1–79:26. https://doi.org/10.1145/3133903

Stephanie Weirich. 2014. Depending on Types. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14).* Association for Computing Machinery, New York, NY, USA, 241. https://doi.org/10.1145/2628136.2631168