Rooting for Efficiency

Mechanised Reasoning about Array-Based Trees in Separation Logic

Qiyuan Zhao
National University of Singapore
Singapore
qiyuanz@comp.nus.edu.sg

George Pîrlea
National University of Singapore
Singapore
gpirlea@comp.nus.edu.sg

Zhendong Ang
National University of Singapore
Singapore
zhendong.ang@u.nus.edu

Umang Mathur
National University of Singapore
Singapore
umathur@comp.nus.edu.sg

Ilya Sergey
National University of Singapore
Singapore
ilya@nus.edu.sg

Abstract

Array-based encodings of tree structures are often preferable to linked or abstract data type-based representations for efficiency reasons. Compared to the more traditional encodings, array-based trees do not immediately offer convenient induction principles, and the programs that manipulate them often implement traversals non-recursively, requiring complex loop invariants for their correctness proofs.

In this work, we provide a set of definitions, lemmas, and reasoning principles that streamline proofs about array-based trees and programs that work with them. We showcase our proof techniques via a series of small but characteristic examples, culminating with a large case study: verification of a C implementation of a recently published *tree clock* data structure in a Separation Logic embedded into Coq.

CCS Concepts: • **Software and its engineering → Software verification**.

Keywords: array-based trees, logical clocks, separation logic

ACM Reference Format:
Qiyuan Zhao, George Pîrlea, Zhendong Ang, Umang Mathur, and Ilya Sergey. 2024. Rooting for Efficiency: Mechanised Reasoning about Array-Based Trees in Separation Logic. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24), January 15–16, 2024, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636501.3636944

1 Introduction

There are hardly any other basic data structures in Computer Science as ubiquitous, versatile, and beloved as *trees*. Used to represent virtually any kind of data with hierarchical ordering, trees admit a simple encoding in both functional programming languages, as algebraic data types (ADT), and in imperative ones, as pointer-based linked structures without internal sharing. Most tree-manipulating programs can be expressed as recursive traversals, whose control flow mimics the shape of the underlying data structure. Thanks to this fact, reasoning about tree manipulations can be conducted via simple induction principles, which made such computations popular topics in studies on program derivation [13, 29], transformation [4, 35], and verification [2, 10].

In this work, we focus on a less well-studied way to represent trees in heap-manipulating programs written in imperative languages such as C: as *arrays*. Developers choose an array-based encoding of trees for efficiency reasons—it allows constant-time random access to node data, requires less memory to store, and enjoys better cache locality than pointer-based representations. From the perspective of formal reasoning about tree-manipulating programs, the array representation comes with a number of new challenges. First of all, tree traversals implemented by means of addressing elements in an array via integer indices do not immediately yield familiar induction principles. Furthermore, programs working with array-based tree representations often exploit the fact that children of a node are arranged into a contiguous array segment: this enables efficient non-recursive traversals, but complicates verification due to the need to devise complex loop invariants. Finally, while arrays are conceptually similar to pointers, they require slightly more delicate reasoning in common formalisms, such as Separation Logic (SL) [28, 31], as proof obligations involving their element indices require one to keep track of the indices' numeric properties to avoid, in particular, out-of-bounds errors.

The motivation for this work came out of our effort of verifying in Coq a C implementation of *Tree Clocks* [22]—an intricate imperative data structure that implements a state-of-the-art version of logical clocks via an array-based tree and extensively takes advantage of the array encoding for the sake of efficiency. SL-based reasoning about complex linked

CPP '24, January 15–16, 2024, London, UK

ACM ISBN 979-8-4007-0488-8/24/01
https://doi.org/10.1145/3636501.3636944

structures with "deep" internal sharing and unstructured aliasing has been addressed, to some extent, in the past [24, 36]. However, we found those developments inapplicable for our goal due to their focus on a more general setup targeting *graph-like* structures and, therefore, imposing extra proof overhead when tackling *tree-specific* proof obligations.

To document the lessons we learned from our verification experience, in this paper, we articulate a number of challenges we faced during while verifying several characteristic procedures that manipulate array-based trees. We then describe reasoning principles and auxiliary definitions encoded in Coq that came in handy when constructing such proofs. In particular, we argue that there are two kinds of representation predicates in Separation Logic for array-based trees (*i.e.*, the *array view* ones and the *tree view* ones), each having its specialised utility in the proof; we also present lemmas for smoothly "switching" between the dual views. Another emphasis of our demonstration is the formulation of an extensible loop invariant for non-recursive tree traversals through a small collection of neat functions defined on the mathematical model of array-based trees. Finally, we sketch the key points of our mechanised correctness proof of the tree clock data structure that has been implemented in C and verified using the Verified Software Toolchain (VST) framework [1], which embeds Separation Logic into Coq.

In summary, this work makes the following contributions:

- A collection of small case studies illustrating the key challenges of reasoning about imperative C programs that manipulate array-based trees (Sec. 2);
- Arboreta: a library of predicates and accompanying higher-order lemmas, as well as a set of principles that streamline the common reasoning patterns for such program (Sec. 3);
- A mechanically verified implementation of Mathur *et al.*'s array-based tree clock data structure in C [22] (Sec. 4).

The code repository for this paper is publicly available at

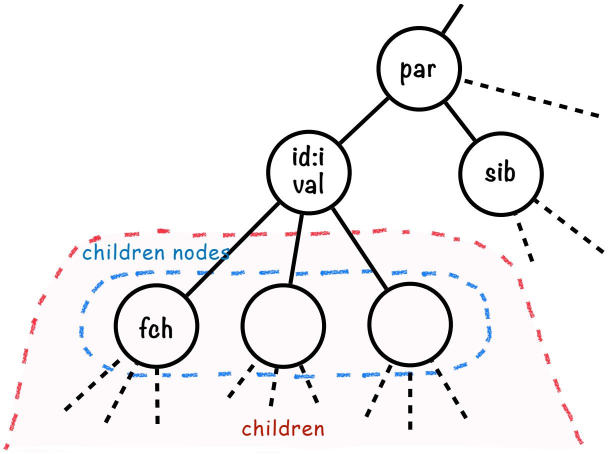
https://github.com/verse-lab/arboreta

2 Overview and Key Ideas

In this work, we reason primarily about *rooted labelled trees* (RLTs), and specifically those implemented using arrays. An RLT is a tree in which each node, besides the data it holds, is given a unique *identifier* (label), usually a natural number. These identifiers allow for node indexing and thereby enable random-access data retrieval from the "carrier" array of the tree. In an imperative programming language like C, a common approach to implementing RLTs is to use an array of **struct**s, where each **struct** contains the information associated with a node, and use array indices as node identifiers. Fig. 1a shows one its possible encoding in C: the tree is represented by the array `tree`, in which `tree[i]` stores the node with identifier *i* (hereafter referred to as node *i*). Each node also stores some data in its `val` field, and its relations to other nodes in the RLT are represented by the fields `par`, `sib`, and `fch` that store the identifiers of the node's parent, right sibling, and first (left-most) child, respectively. Fig. 1b gives a visual depiction of such a tree. As indicated in Fig. 1b, this structure allows the representation of RLTs of any arity.

```
struct node { int val, par, sib, fch; };
struct node tree[N];
```

(a) RLT encoding in C.



(b) Correspondence between a tree node and the **struct** fields.

Fig. 1. RLT: encoding in C (top) and its depiction (bottom).

In the rest of this section, we provide a series of illustrative examples of programs that manipulate RLTs encoded as shown in Fig. 1a, articulating the challenges of structuring formal verification of such programs.

Formalising RLTs. Fig. 2 shows a natural encoding of an RLT in Coq as an inductive algebraic data type: an RLT is constructed with the `Node` constructor, which carries the node identifier `id`, the node information `val` and the list `chn` of children. Even though the definition of the `tree` data type offers no constructor for an empty tree, leafs (*i.e.*, nodes without children) can be modelled as instances of the `Node` constructor with empty `chn`. To ensure that the nodes in a tree *tr* all have *distinct* identifiers, we use an extrinsic predicate NoDupId(tr), whose presentation is postponed till Sec. 3.1.

While this fairly standard inductive definition is sufficient for modelling the tree structure and stating and proving mathematical properties of purely-functional RLTs, it is not immediately suitable for specifying and verifying imperative programs that manipulate array-based RLTs: the *logical* tree structure is not directly related to the *physical* one in memory. To bridge this gap, we employ a standard technique of defining a *representation predicate* in Separation Logic [6, 7, 34] that relates the algebraic definition of RLTs in Coq to the memory layout of the RLT in C.

To define a representation predicate $\text{tree\_rep}_{\text{arr}}$ for array-based RLTs, we can use the conventional predicate $\text{arr}(p, \ell)$

```
Inductive tree: Type := Node (id val: nat) (chn: list tree).
Definition NoDupId (tr: tree): Prop := (* elided *)
```

Fig. 2. An algebraic encoding of RLT in Coq.

for arrays (available in many existing Coq embeddings of SL [1, 8]), which states that an address p in memory is a *base pointer* of an array whose elements are the elements of the (mathematical) list ℓ. Concretely, we make ℓ into a *list of payloads*, where each payload is the functional model of a single instance of the `node` structure shown in Fig. 1a.

Unlike the C definition in Fig. 1a, in the Coq encoding, the child subtrees of a node are represented as a finite list. The challenge in relating the C encoding, which connects a node to its parents/children/sibling by virtue of the indices represented by integer fields, with the Coq encoding from Fig. 2 lies in recovering the list of nodes from the inductive tree definition, and specifically in constructing the `par`, `sib`, and `fch` fields to express the tree structure. To this end, we define a recursive *projection* predicate treelist_proj that relates a list of trees to a list of payloads:

$$
\begin{aligned}
&\mathrm{treelist\_proj}(\mathrm{nil}, par, \ell) \;\triangleq\; \top\\
&\mathrm{treelist\_proj}(\mathrm{Node}(id, val, chn) :: trs, par, \ell) \;\triangleq\\
&\quad \ell[id] = \mathrm{payload}(val, par, \mathrm{hid}(trs), \mathrm{hid}(chn)) \land\\
&\quad \mathrm{treelist\_proj}(chn, id, \ell) \land \mathrm{treelist\_proj}(trs, par, \ell)
\end{aligned}
$$

As per the definition above, the projection predicate takes as input a *list of trees*, rather than a single tree, in order to match the structure of the Coq definition *wrt.* children. For $\mathrm{Node}(id, val, chn)$ in the list, the predicate asserts that its payload is the id^{th} element of ℓ (denoted $\ell[id]$), with the appropriate parent. The right sibling and first child are obtained by retrieving the identifier of the heading node (hid) in the trs and chn lists, respectively. The remainder of the payload list ℓ is constrained by two recursive applications of the predicate, $\mathrm{treelist\_proj}(chn, id, \ell)$ and $\mathrm{treelist\_proj}(trs, par, \ell)$. The indices of the nodes in the payload list ℓ are the same as their identifiers in the definition from Fig. 2, but treelist_proj could assign identifiers arbitrarily, as long as they are distinct.

With the above predicate, we can now define a projection on trees (rather than list of trees) and the representation predicate for array-based RLTs in terms of Separation Logic:

$$
\begin{aligned}
\mathrm{tree\_proj}(tr, \ell) &\triangleq \mathrm{treelist\_proj}(tr :: \mathrm{nil}, -1, \ell)\\
\mathrm{tree\_rep}_{\mathrm{arr}}(p_{\mathtt{tree}}, tr) &\triangleq \exists \ell, \lceil \mathrm{tree\_proj}(tr, \ell) \rceil * \mathrm{arr}(p_{\mathtt{tree}}, \ell)
\end{aligned}
\tag{1}
$$

where $\mathrm{tree\_rep}_{\mathrm{arr}}(p_{\mathtt{tree}}, tr)$ represents the array-based RLT tr whose base pointer is $p_{\mathtt{tree}}$ (*e.g.*, the `tree` pointer in Fig. 1a). As customary in SL, the $*$ connective stands for the *separating conjunction* [31], and $\lceil \ldots \rceil$ denotes pure assertions.

2.1 Specifying Computations with Array-Based RLTs

Let us now illustrate the representation predicate by specifying and verifying a couple of RLT-manipulating programs.

```
Definition isSome [A : Type] (x : option A) :=
  match x with Some _ => true | None => false end.

Fixpoint find_val (x : nat) (tr : tree) : option nat :=
  let 'Node id v chn := tr in
  if (x =? id)%nat then Some v
  else match find isSome (map (find_val x) chn) with
       | Some res => res
       | None => None
       end.
```

Fig. 3. Coq definitions for retrieving an value from RLT.

Example 1: Random Access. Consider the following function that returns a value of an RLT node given its index `x`:

```
int get_val (int x) { return tree[x].val; }
```

Ascribing a good formal specification to this one-liner requires one to provide a couple of auxiliary definitions that amount to several lines of Coq and are shown in Fig. 3. Using the Coq function find_val as a helper, we can now ascribe the following Hoare triple to `get_val`:

$$
\begin{gathered}
\{\mathrm{tree\_rep}_{\mathrm{arr}}(p_{\mathtt{tree}}, tr) * \lceil \mathrm{find\_val}(x, tr) = \mathrm{Some}(v) \rceil\}\\
\mathtt{get\_val}(x)\\
\{\lambda ret.\ \mathrm{tree\_rep}_{\mathrm{arr}}(p_{\mathtt{tree}}, tr) * \lceil ret = v \rceil\}
\end{gathered}
\tag{2}
$$

The specification above states that, assuming the node with the identifier x holds the value v in the algebraic representation of the RLT tr and the array-based representation of tr is available to `get_val`, the result ret of the call will be exactly v. To establish this specification, we first need to prove an important property of find_val, namely, that if $\mathrm{find\_val}(x, tr) = \mathrm{Some}(v)$, then for any ℓ that satisfies $\mathrm{tree\_proj}(tr, \ell)$, $\ell[x]$ contains v. This fact can be proven by induction on the algebraic tree tr via the following induction principle stating that a property holds on an RLT if it holds on all children and on the root node itself:[1]

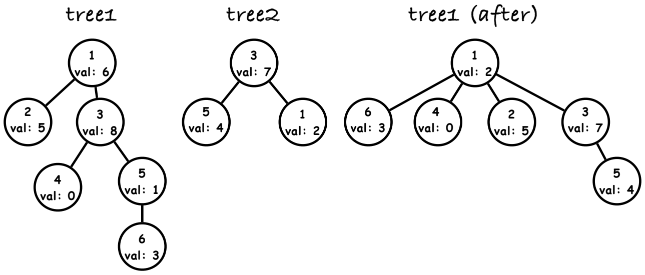
```
Lemma tree_ind' (P: tree -> Prop)
  (Hindstep : forall (id v: nat) (chn: list tree),
    (forall ch, In ch chn -> P ch) -> P (Node id v chn)):
  forall (tr: tree), P tr.
```

The specification (2) can be then proven using standard Separation Logic rules. First, $\mathrm{tree\_rep}_{\mathrm{arr}}(p_{\mathtt{tree}}, tr)$ gives us the logical list ℓ, which we pass to the find_val property, obtaining the fact that $\ell[x]$ contains v. Next, from $\mathrm{arr}(p_{\mathtt{tree}}, \ell)$ we know that `tree[x]` stores the equivalent of the payload $\ell[x]$. Finally, from the above, we conclude that `tree[x].val` returns $\ell[x]$`.val`. The key enabler of this proof is the representation predicate that links the algebraic tree with the array-based one. Random access operations similar to `get_val` are paramount in implementations that manipulate array-based RLTs, and we make use of such specifications in our chief case study with tree clocks outlined in Sec. 4.

[1] Coq automatically derives a weaker induction principle for the `tree` from Fig. 2, hence we define `tree_ind'` and prove it as a standalone theorem.

```
struct node tree1[N], tree2[N]; // trees
int stack[N];                   // assisting stack
void copyval_and_move (int root1, int root2) {
  // root1: the root of tree1
  // root2: the root of tree2
  int top = -1;
  stack[++top] = root2;
  while (top >= 0){
    int x = stack[top--];
    tree1[x].val = tree2[x].val;
    if (tree1[x].fch != -1)
      move_first_child(root1, x); // defined in Fig. 5
    int tmp = tree2[x].fch;
    while (tmp != -1){
      stack[++top] = tmp;
      tmp = tree2[tmp].sib;
    }
  }
}
```

(a) Structure-changing non-recursive tree traversal.



(b) Example of executing copyval_and_move.

Fig. 4. A structure-changing traversal and its depiction.

Example 2: Structure-Changing Tree Traversals. Realistic programs that manipulate RLTs can be quite complex, and may perform (imperative, non-recursive) *traversals* interspersed with *structure-changing* operations.[2] Fig. 4a shows an example of such a program, which (i) traverses the tree stored in tree2, (ii) copies the val of each traversed node x to the node with the same identifier (the "updated node") in the tree stored in tree1, and (iii) moves the first child of the updated node to be the first child of the node root1. Fig. 4b shows an example execution of copyval_and_move. In the resulting tree, the value of node 3 has been updated to 7, and node 3's first child in tree1, *i.e.*, node 4, has been moved to become a child node of node 1, which is the root of tree1. Similarly, node 5's value has been updated to 4, and node 5's first child, *i.e.*, node 6, has been attached to node 1. Finally, the value of node 1 is updated to 2. This example is a simplified version of the manipulations that take place in tree clocks, and we dedicate the next two subsections to zooming in on its two challenging aspects: changing the tree structure and performing its non-recursive traversal.

```
void move_first_child (int dst, int src) {
  /* precondition: tree1[src].fch != -1 and
     node dst is not in tree1[src].fch */
  int tmp = tree1[src].fch;
  tree1[src].fch = tree1[tmp].sib;
  tree1[tmp].sib = tree1[dst].fch;
  tree1[dst].fch = tmp;
  tree1[tmp].par = dst;
}
```

Fig. 5. Example program that changes the tree's structure.

2.2 Structure-Changing Tree Operations

Before we look into copyval_and_move, let us first verify its subprocedure move_first_child, shown in Fig. 5.

This procedure makes the first child of node src become the first child of node dst, while keeping the rest of the tree's structure unchanged. To give its specification, we can (a) write one or more recursive functions in Coq exhibiting the same behaviour as the imperative program (*e.g.*, one for popping a child of node i and another for prepending a child), and then (b) state that the logical tree obtained after applying these functions corresponds to the tree in the postcondition:

$$\begin{gathered}\{\mathrm{tree\_rep_{arr}}(p_{\texttt{tree}}, tr) * \lceil\cdots\rceil\} \\ \texttt{move\_first\_child}(dst, src) \\ \{\lambda\_.\mathrm{tree\_rep_{arr}}(p_{\texttt{tree}}, tr')\}\end{gathered} \tag{3}$$

where tr' in the postcondition may be similar in form to $\mathrm{prepend\_child}(\mathrm{pop\_child}(tr, src), dst)$. We give one possible definition of prepend_child later in Sec. 3.2.4.

To prove the specification (3), first observe that this program only involves reads and writes on the fields of the array tree. We can unfold the $\mathrm{tree\_rep_{arr}}$ in the precondition to obtain the initial payload list ℓ by existential elimination. Similarly, in the postcondition, the resulting tree is represented by another payload list ℓ' such that $\mathrm{arr}(p_{\texttt{tree}}, \ell')$ holds, and we know that ℓ' is obtained from ℓ by applying a sequence of transformations. It then suffices to show that $\mathrm{tree\_proj}(tr', \ell')$ holds. Unfortunately, this proof obligation can be *very* cumbersome, since we need to show that the payloads not touched by move_first_child have remained unchanged in the process. We can prove this by induction, but it is tedious, and the proof would need to be repeated anew for every different structure-changing operation.

Key Idea: Dual Views. What we would really want are some *localised* reasoning principles that would allow us to prove $\mathrm{tree\_proj}(tr', \ell')$ by only requiring reasoning about the modified part of the payloads and the tree. However, this is hard to achieve with the current representation predicate.

Prior work on Separation Logic-based verification has shown that one can get exactly this workflow by defining an *alternative* representation predicate that reflects tree structure via suitable placed usages of the separating conjunction in its definition [6, 7]. We will refer to such a predicate that

[2] For ease of presentation, all non-recursive traversals in this paper visit child nodes from last to first. Regular traversals can be handled analogously.

allows one to reason about the memory manipulations *inductively* and employ proper localised reasoning rules (*e.g.*, Frame), as a *tree view* predicate. In Sec. 3.2, we will show how to define such a tree view predicate, which facilitates verifying structure-changing RLT operations. At the same time, we will keep using the previously-defined *array view* predicate to deal with random access operations, and will be switching between the two views as needed to use the one best suited for the verification task at hand.

2.3 Non-Recursive Tree Traversals

Preorder or postorder tree traversals are easy to implement recursively in both functional and imperative styles. Since RLT nodes have unique identifiers, we can also implement *non-recursive* traversals using a stack. This avoids recursion overhead (*e.g.*, allocation of stack frames), and is therefore preferred in scenarios where efficiency is key.

The difficulty with certifying such traversals lies in stating the loop invariant. To illustrate this, let us temporarily turn our attention from *loop-based* implementation of `copyval_and_move` to a simpler program in Fig. 6 that performs a *recursive* preorder traversal on array-based RLTs. The program computes the maximum value `val` in the tree, and we can straightforwardly define the functional model of this imperative program in Coq:

```
Fixpoint max_val (tr : tree) : nat :=
  let 'Node _ v chn := tr in
  Nat.max v (list_max (map max_val chn)).
```

Since `max_val_rec` performs recursion nearly identically to `max_val`, a feasible SL specification for `max_val_rec` is:

$$\begin{gathered}\{\mathsf{tree\_rep}'_{\mathsf{arr}}(p_{\mathtt{tree}}, tr, par)\}\\ \mathtt{max\_val\_rec}(\mathsf{id\_of}(tr))\\ \{\lambda ret.\ \mathsf{tree\_rep}'_{\mathsf{arr}}(p_{\mathtt{tree}}, tr, par) * \lceil ret = \mathsf{max\_val}(tr)\rceil\}\end{gathered} \tag{4}$$

where $\mathsf{id\_of}(tr)$ returns the identifier of tr's root, and

$$\begin{aligned}\mathsf{tree\_rep}'_{\mathsf{arr}}(p_{\mathtt{tree}}, tr, par) \ &\triangleq\\ &\exists \ell, \lceil \mathsf{treelist\_proj}(tr :: \mathsf{nil}, par, \ell)\rceil * \mathsf{arr}(p_{\mathtt{tree}}, \ell).\end{aligned}$$

We have to use the generalised representation predicate above because `par` is not -1 in the recursive call.

We can now establish specification (4) by (a) showing that $\mathsf{tree\_rep}'_{\mathsf{arr}}(p_{\mathtt{tree}}, \mathsf{Node}(id, val, chn), par)$ entails that the predicate $\mathsf{tree\_rep}'_{\mathsf{arr}}(p_{\mathtt{tree}}, ch, id)$ holds for all $ch \in chn$, and (b) using "the value of `maxi` is the maximum value amongst tr's root and some prefix of its children" as a loop invariant.

Although, as we have seen, a recursive traversal similar to `max_val_rec` is straightforward to specify and prove, the situation is quite different for `copyval_and_move`. Specifically, it is unclear how to specify the loop invariant for its outer **`while`**-loop. There are two aspects we need to capture in that loop invariant: (a) the contents of the `stack` and (b) the visited nodes (*i.e.*, the nodes that have already been popped from the stack in previous iterations). The two must be consistent (*e.g.*, a visited node should not appear in the stack) and maintained synchronously (*e.g.*, once a node is visited in the current iteration, its children nodes should be pushed into the stack). The difficulty lies in both characterising these two aspects and describing how they evolve in a traversal. In some cases, it might suffice to define a ghost state for the visited nodes, but this does not generalise for arbitrary traversals.

```
int max_val_rec (int rt) {
  int maxi = tree[rt].val;
  int tmp = tree[rt].fch;
  while (tmp != -1){
    int res = max_val_rec(tmp);
    if (maxi < res) maxi = res;
    tmp = tree[tmp].sib;
  }
  return maxi;
}
```

Fig. 6. Recursive traversal of an array-based RLT.

Key Idea: Tree Splitting. To verify non-recursive traversals, we need a precise characterisation of the tree structure that has been already visited. This can be achieved by "splitting" the set of array-hosted nodes of the tree into those already visited and those that yet remain to be processed. Luckily, the spatial structure of the array-based RLTs makes it possible to define such invariants by instantiating a common "template" that constrains the two subsets of the nodes. We detail this technique Sec. 3.3, providing a description of the helper lemmas that facilitate such proofs.

3 Arboreta: Proofs about Array-Based Trees

In this section, we elucidate the design of Arboreta—a Coq library with a set of proof principles that facilitate mechanised reasoning about array-based tree manipulations in Separation Logic. We introduce its fundamental components in Sec. 3.1. In Sections 3.2 and 3.3, we develop and apply specialised reasoning principles to solve the challenges outlined previously in Sections 2.2 and 2.3, respectively.

3.1 Reasoning Principles for Rooted Trees

Core to Arboreta is Arboreta-P, a small collection of useful definitions and lemmas for *pure* reasoning about rooted (unlabelled and labelled) trees. The library provides a definition of *generic* rooted trees, parametrised by a type parameter `A`, which is the type of node data, shown in Fig. 7a. For convenience, we keep using `tree` as the type name of trees.

Expansion. One of the core definitions is the *expansion* function, which, when applied to an algebraic tree `tr`, returns a list containing all the subtrees of `tr`. Fig. 7b shows its definition, and three derived definitions, for the size of a tree, the subtree relation, and the list of node data. Since a bijection exists between nodes and subtrees, expansion serves as a building block for other definitions, *e.g.*, `NoDupId`.

```
Parameter A : Type.
Inductive tree : Type := Node (a : A) (chn : list tree).
Definition data_of tr := let 'Node a _ := tr in a.
Definition chn_of tr := let 'Node _ chn := tr in chn.
```

(a) Rooted trees and getters.

```
Fixpoint expand tr : list tree :=
  let 'Node a chn := tr in tr :: (flat_map expand chn).
Definition size tr : nat := length (expand tr).
Definition subtr tr1 tr2 : Prop := In tr1 (expand tr2).
Definition data_list_of tr : list A := map data_of (expand tr).
```

(b) Expansion and derived definitions.

```
Parameters (B : Type) (id_of_data : A -> B).
Definition id_of tr : B := id_of_data (data_of tr).
Definition id_list_of tr : list B := map id_of (expand tr).
Definition NoDupId tr : Prop := NoDup (id_list_of tr).
```

(c) Definitions related to node identifiers.

```
Parameter (B_eqb : B -> B -> bool).
Definition has_same_id x tr := B_eqb (id_of tr) x.
Definition find_node x tr : option tree :=
           find (has_same_id x) (expand tr).
```

(d) Definition of node finding function. B_eqb is a binary boolean function for checking whether two terms of type B are equal or not.

```
Fixpoint locate tr pos : option tree := (* ... *)
Fixpoint locate_update tr (pos : list nat) new : tree :=
  match pos with
  | nil => new
  | x :: pos' => let 'Node a chn := tr in
    match nth_error chn x with
    | Some ch =>
      Node a (upd_nth x chn (locate_update ch pos' new))
    | None => tr
    end
  end.
```

(e) Using node coordinates to update subtrees. $\mathrm{nth\_error}(\ell, x)$ returns $\mathrm{Some}(\ell[x])$ if $x < |\ell|$ and None otherwise. $\mathrm{upd\_nth}(x, \ell, a)$ returns the resulting list obtained by updating $\ell[x]$ to a.

```
Inductive prefix : tree -> tree -> Prop :=
  prefix_intro : forall a chn chn_sub prefix_chn,
    subseq chn_sub chn ->
    Forall2 prefix prefix_chn chn_sub ->
    prefix (Node a prefix_chn) (Node a chn).
Lemma prefix_data_is_subset tr1 tr2 (Hpref : prefix tr1 tr2):
  subseq (data_list_of tr1) (data_list_of tr2).
```

(f) Definition of tree prefix and one of its property. $\mathrm{Forall2}(P, \ell_1, \ell_2)$ holds iff $|\ell_1| = |\ell_2|$ and $\forall i, P(\ell_1[i], \ell_2[i])$. $\mathrm{subseq}(\ell_1, \ell_2)$ holds iff ℓ_1 is a subsequence of ℓ_2.

Fig. 7. Main Coq definitions from Arboreta-P.

Node identifiers. As the type of node data is parameterised, we also parameterise the type of node identifiers into B, as shown in Fig. 7c. id_of_data extracts the node identifier from a node's data, and id_of extracts it from a node. NoDupId is defined in terms of expansion and id_of.

Finding by identifier. When identifiers come equipped with decidable equality (*e.g.*, natural numbers), Arboreta-P provides a function find_node(x, tr) for finding the subtree of tr whose root has identifier x, as shown in Fig. 7d.

Node coordinate. We can assign a unique *coordinate* to each node in the tree. A coordinate or *position* describes how to find that node from the root in a top-down manner, and can be represented in Coq as a list of natural numbers storing a child index to visit at each level. We define a locate function to access a node by coordinate (the definition is trivial and elided), and a locate_update function to replace the subtree at a given coordinate, as shown in Fig. 7e.

Tree prefix. We say that a rooted tree tr_1 is a *prefix* of tr_2 if tr_1 is a subgraph of tr_2 and both trees have the same root. This definition is encoded by the inductive predicate from Fig. 7f, along with a custom induction principle (not shown). With such an abstract definition, we can prove properties of prefixes in Arboreta-P, and then prove that the trees returned by the (possibly complicated) functions we want to verify are prefixes. For instance, we use the prefix_data_is_subset property in the tree clock case study (*cf.* Sec. 4).

3.2 Dual Views: From Array to Tree and Back Again

Recall that in Sec. 2.2, we encountered the need to reason about trees in a structure-aware fashion, and envisioned an inductively defined *tree view* predicate to be used as an alternative to the *array view*, in order to reason with ease about structure-changing operations. In this section, we (a) derive the tree view predicate from the array view one, (b) exploit the tree view to apply local reasoning principles, and finally (c) shift back to the array view. In other words, we work through the (general) way to apply the following rule of consequence, and showcase it on move_first_child from Fig. 5:

$$\frac{P_{\mathsf{arr}} \vdash P_{\mathsf{tree}} \qquad \{P_{\mathsf{tree}}\}\ c\ \{Q_{\mathsf{tree}}\} \qquad Q_{\mathsf{tree}} \vdash Q_{\mathsf{arr}}}{\{P_{\mathsf{arr}}\}\ c\ \{Q_{\mathsf{arr}}\}}$$

3.2.1 From Array View to Tree View. As a reminder, the representation predicate of an array is typically defined as a collection of contiguous memory blocks, as follows:[3]

$$\mathrm{arr}(p, \ell) \triangleq \mathop{\Large *}_{i \in [0, |\ell|)} p + i \times \mathrm{sizeof}(T) \mapsto \ell[i] \qquad (\textsc{ArrDef})$$

where T is the type of the payload and $\mathrm{sizeof}(T)$ denotes how much space the payload occupies in memory. Since the separating conjunction $*$ is commutative and associative, we can reorganise the memory blocks in the array according

[3] In practical SL frameworks, the right-hand side may come with side-conditions, such as p != NULL. We omit them to simplify the presentation.

to the tree's structure, and inductively define the tree view predicate $\mathsf{tree\_rep}_{\mathsf{tree}}$ as follows:

$$\begin{aligned}
&\mathsf{treelist\_rep}_{\mathsf{tree}}(\mathsf{nil}, par, p) \triangleq \mathsf{emp}\\
&\mathsf{treelist\_rep}_{\mathsf{tree}}(\mathsf{Node}(id, val, chn) :: trs, par, p) \triangleq\\
&\quad p + id \times \mathsf{sizeof}(T) \mapsto \mathsf{payload}(val, par, \mathsf{hid}(trs), \mathsf{hid}(chn)) \ast\\
&\quad \mathsf{treelist\_rep}_{\mathsf{tree}}(chn, id, p) \ast \mathsf{treelist\_rep}_{\mathsf{tree}}(trs, par, p)\\
&\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr) \triangleq \mathsf{treelist\_rep}_{\mathsf{tree}}(tr :: \mathsf{nil}, -1, p)
\end{aligned}$$

where hid is defined in Sec. 2 before Sec. 2.1. Noting the similarity to the previously defined list-based $\mathsf{tree\_rep}_{\mathsf{arr}}$ (1), we can prove the following entailment by induction on tr:

$$\begin{aligned}
&\mathop{\ast}_{i \in \mathsf{ids}(tr)} p + i \times \mathsf{sizeof}(T) \ast \lceil \mathsf{tree\_proj}(tr, \ell) \rceil\\
&\qquad \vdash \mathsf{tree\_rep}_{\mathsf{tree}}(p, tr)
\end{aligned} \tag{MemBlksToTree}$$

where ids is a shorthand of `id_list_of` from Fig. 7c. Therefore, given $\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr)$, we can obtain the tree representation $\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr)$ by first unfolding arr by ArrDef and then apply MemBlksToTree. We then combine the two steps together and get the following:

$$\begin{aligned}
&\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) \vdash\\
&\mathop{\ast}_{i \in \mathsf{rem}(tr, |\ell|)} p + i \times \mathsf{sizeof}(T) \mapsto \ell[i] \ast \mathsf{tree\_rep}_{\mathsf{tree}}(p, tr)
\end{aligned} \tag{ArrToTree}$$

where $\mathsf{rem}(tr, |\ell|) \triangleq [0, |\ell|) \setminus \mathsf{ids}(tr)$ contains the "remaining" memory blocks, *i.e.*, the allocated array indices not used by the tree. If the size of tr is $|\ell|$, *i.e.*, $\mathsf{rem}(tr, |\ell|)$ is empty, ArrToTree simplifies to $\mathsf{tree\_rep}_{\mathsf{arr}}(p, tr) \vdash \mathsf{tree\_rep}_{\mathsf{tree}}(p, tr)$.

3.2.2 Local Reasoning with Tree View.

Let us define the predicate $\mathsf{tree\_rep}'_{\mathsf{tree}}$ in a way similar to $\mathsf{tree\_rep}'_{\mathsf{arr}}$ (4):

$$\begin{aligned}
&\mathsf{tree\_rep}'_{\mathsf{tree}}(p, \mathsf{Node}(id, val, chn), par, sib) \triangleq\\
&\quad p + id \times \mathsf{sizeof}(T) \mapsto \mathsf{payload}(val, par, sib, \mathsf{hid}(chn)) \ast\\
&\quad \mathsf{treelist\_rep}_{\mathsf{tree}}(chn, id, p)
\end{aligned} \tag{5}$$

Using the properties of the magic wand [6], we can reflect the modifications to the functional tree made by locate_update (Fig. 7e) onto the heap state, expressing this by the following entailment, which can be proved by induction on pos:

$$\begin{aligned}
&\lceil \mathsf{locate}(tr, pos) = \mathsf{Some}(sub) \rceil \ast \mathsf{tree\_rep}'_{\mathsf{tree}}(p, tr, par, sib)\\
&\vdash\\
&\exists par', sib',\ \mathsf{tree\_rep}'_{\mathsf{tree}}(p, sub, par', sib') \ast\\
&\left(\begin{array}{c} \forall sub', \mathsf{id\_of}(sub) = \mathsf{id\_of}(sub') \implies \\ \left(\begin{array}{c} \mathsf{tree\_rep}'_{\mathsf{tree}}(p, sub', par', sib') \mathrel{-\!\!\ast} \\ \mathsf{tree\_rep}'_{\mathsf{tree}}(p, \mathsf{locate\_update}(tr, pos, sub'), par, sib) \end{array} \right) \end{array} \right)
\end{aligned} \tag{WandFrameUpdate}$$

The entailment in WandFrameUpdate might look a bit intimidating, but the only thing it does is instantiating the analogue of the "modus ponens" rule for $\ast / \mathrel{-\!\!\ast}$, "pulling out" the $\mathsf{tree\_rep}'_{\mathsf{tree}}(p, sub', par', sib')$ assertion. Note that WandFrameUpdate allows "pulling out" only a single subtree at a time, which is nevertheless sufficient for our verification task about tree clock. We will discuss this limitation in Sec. 5.

3.2.3 From Tree View to Array View.

Thus far, we have defined the tree view predicate and shown how to obtain it from the array view predicate. However, in practice, the array view predicate is independently useful for reasoning about read-only operations, especially random array accesses. This utility may stem from specialised support provided by verification tools for handling array operations, for instance. To this end, a natural question could be whether we are able to switch between the array view and the tree view on demand so as to enjoy the best parts of both views.

The answer is affirmative. From the definition ArrDef, we can prove the following entailment, which "reconstructs" an array from contiguous memory blocks, by induction on n. Here, n is a natural number and the type of f is $\mathbb{N} \to T$.

$$\begin{aligned}
&\mathop{\ast}_{i \in [0,n)} p + i \times \mathsf{sizeof}(T) \mapsto f(i) \vdash\\
&\exists \ell,\ \lceil (\forall i \in [0, n), \ell[i] = f(i)) \wedge |\ell| = n \rceil \ast \mathsf{arr}(p, \ell)
\end{aligned} \tag{ArrIntro}$$

And since the tree view predicate can be also regarded as a bunch of memory blocks, we can prove by induction on tr that it can be "shattered" into memory blocks whose content is specified by a function from identifiers to payload:

$$\begin{aligned}
&\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr) \vdash\\
&\exists f,\ \lceil \forall \ell, (\forall i \in \mathsf{ids}(tr), \ell[i] = f(i)) \implies \mathsf{tree\_proj}(tr, \ell) \rceil \ast\\
&\mathop{\ast}_{i \in \mathsf{ids}(tr)} p + i \times \mathsf{sizeof}(T) \mapsto f(i)
\end{aligned} \tag{TreeToMemBlks}$$

Finally, by gathering the payloads with indices in $\mathsf{rem}(tr, |\ell|)$, we can reconstruct the array view from the tree view:

$$\begin{aligned}
&\mathop{\ast}_{i \in \mathsf{rem}(tr, |\ell|)} p + i \times \mathsf{sizeof}(T) \mapsto f(i) \ast \mathsf{tree\_rep}_{\mathsf{tree}}(p, tr)\\
&\qquad \vdash \mathsf{tree\_rep}_{\mathsf{arr}}(p, tr)
\end{aligned} \tag{TreeToArr}$$

Again, once $\mathsf{rem}(tr, |\ell|)$ is empty, TreeToArr simplifies to $\mathsf{tree\_rep}_{\mathsf{tree}}(p, tr) \vdash \mathsf{tree\_rep}_{\mathsf{arr}}(p, tr)$. In this case, we can switch "seamlessly" between the two views using ArrToTree and TreeToArr.

3.2.4 Dual Views in Action.

The ability to switch between the array and tree views allows for relatively straightforward verification of `move_first_child` from Sec. 2.2 against the specification (3). To do so, we first express the local modifications via locate_update. For example, prepending a child ch to a specific node of tr can be implemented as follows, when given the coordinate pos of that node inside tr:

$$\begin{aligned}
&\mathsf{prepend\_child}(tr, ch) \triangleq\\
&\qquad \mathsf{locate\_update}(tr, pos, \mathsf{Node}(a, ch :: chn))
\end{aligned}$$

where we let $\mathsf{locate}(tr, pos)$ be $\mathsf{Some}(\mathsf{Node}(a, chn))$. We can implement popping the first child analogously. After those instantiations, we can then switch from the array view in the precondition to the tree view via ArrToTree, apply WandFrameUpdate to reason about the affected local part, and finally recover the original array view via TreeToArr.

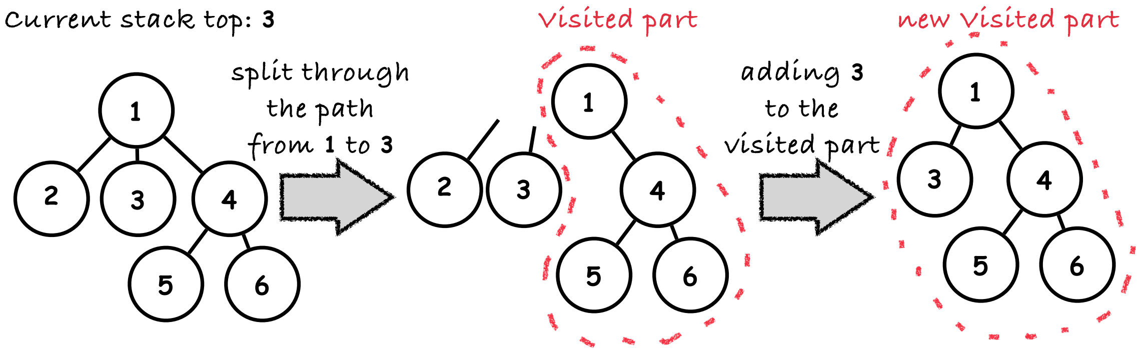


Fig. 8. Splitting a tree vertically *wrt.* the node 3.

3.3 Loop Invariants for Non-Recursive Traversals

As the last component of Arboreta we present an approach for stating loop invariants to verify non-recursive tree traversals similar to that of `copyval_and_move` from Fig. 4a. Recall that the solution hinted at the end of Sec. 2.3 was to explicitly characterise the "visited part" of the tree in the loop invariant, relating it to the contents of the explicit stack.

In such traversals, at the beginning of each iteration of the outer **`while`**-loop, the visited part is likened to the "right half" of the tree obtained by "splitting" the original tree along the path from the root down to the node that will be processed in the current iteration (*i.e.*, the node at the top of the stack; represented by `x` in Fig. 4a, for example), which we refer to as the *stack top node* hereafter. After completing the iteration, the stack top node will belong to the visited part in the loop invariant. This process is depicted in Fig. 8 with regard to the "working" node 3. As we will soon see, a visited part must be a prefix of the original tree (in the sense of the definition from Fig. 7f), so we will refer to it as the *visited prefix*. In addition, we name the visited prefix at the beginning/end of the iteration as the *pre/post-iteration* visited prefix, respectively.

In the remainder of this subsection, we will walk through the intuition of formalising the idea of splitting the tree along the path, showing how it leads to an extensible invariant definition for non-recursive traversals, and demonstrating the utility of that definition to verify the program from Fig. 4a.

3.3.1 Vertical Tree Splitting.

Fig. 9a provides the definition of a function for splitting the tree vertically *wrt.* a given node, which is represented by its position. The function `vsplit(full, tr, pos)` returns the part "on the right" after splitting `tr` *wrt.* the node at `pos`, whose children will be included in the result iff `full` is `true` (see the first branch of **`match`** `pos` **`with`** in the definition of `vsplit`).

We can now use `vsplit` to define post-iteration visited prefix for a traversal loop. As an example, let us take `tr` to be the tree in the leftmost part of Fig. 8. Then the coordinate of the node 3 in `tr` is $\texttt{pos}_3$ = `[1]`, and we can check that the result of `vsplit(false, tr,` $\texttt{pos}_3$`)` is the post-iteration visited prefix shown in the rightmost part of Fig. 8.

```
Fixpoint vsplit (full: bool) tr (pos: list nat) : tree :=
  let 'Node a chn := tr in
  match pos with
  | nil => Node a (if full then chn else nil)
  | x :: pos' =>
    match nth_error chn x with
    | Some ch =>
      let chn' := vsplit full ch pos' :: drop (S x) chn in
      Node a chn'
    | None => Node a nil
    end
  end.

Lemma vsplit_is_prefix : forall full tr pos,
  prefix (vsplit full tr pos) tr.
```

(a) Definition and lemma of vertical tree splitting.

```
Definition post_iter_visited_prefix tr pos : tree :=
  vsplit false tr pos.

Definition rsibpos (pos : list nat) : list nat :=
  match rev pos with nil       => nil
                   | x :: pos' => rev ((S x) :: pos') end.

Lemma vsplit_norsib : forall tr pos, locate tr pos = None ->
  vsplit false tr pos = vsplit false tr (removelast pos).

Definition pre_iter_visited_prefix tr pos : tree :=
  vsplit (isSome (locate tr (rsibpos pos))) tr (rsibpos pos).
```

(b) Definitions of pre/post-iteration visited prefixes. When computing the pre-iteration visited prefix, the value of `full` depends on the existence of the right sibling of the node at `pos`. Note that if the right sibling does not exist, then by `vsplit_norsib`, the computed result will be equal to the post-iteration visited prefix applied to the coordinate of the parent node, namely `(removelast pos)`.

Fig. 9. Coq definitions related with vertical tree splitting.

Even though the function `vsplit` is defined to produce the post-iteration visited prefix, it can also be used to obtain the pre-iteration visited prefix: the pre-iteration visited prefix is exactly the post-iteration visited prefix after "subtracting"

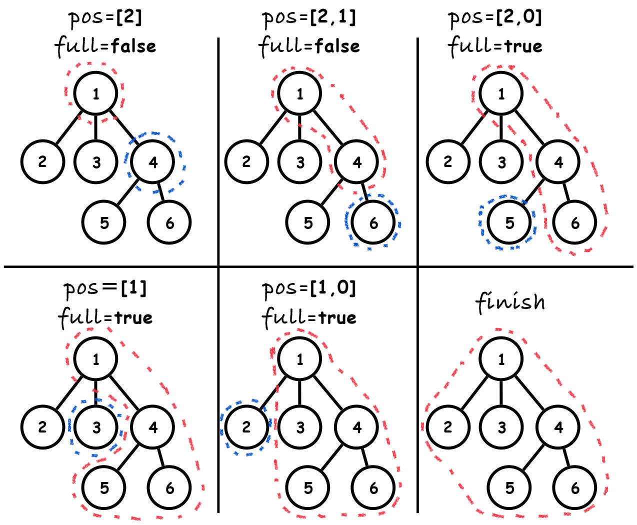


Fig. 10. Tree traversal and vertical splitting.

the stack top node. It turns out that this definition of the pre-iteration visited prefix coincides with the result obtained by `vsplit` using the coordinate of the *right sibling* (or the parent, if the right sibling does not exist) of the stack top node. Moreover, the coordinate of the right sibling of a node can be calculated from the node’s own coordinate via the function `rsibpos` in Fig. 9b.

We summarise the formal definitions of pre/post-iteration visited prefixes in Fig. 9b, from which we know that they are indeed tree prefixes by the lemma `vsplit_is_prefix`. Fig. 10 depicts a sequence of steps showing how the non-recursive traversal progresses. Each subfigure snapshots the state at the start of the corresponding loop iteration. The portion encircled by red dashed line indicates the pre-iteration visited prefix, and a node surrounded by blue dashed line denotes the stack top node with `pos` being its coordinate. Readers are invited to validate the definitions from Fig. 9b by using the `pos` and `full` in each subfigure.

3.3.2 Retrieving Stack Contents.

Perhaps surprisingly, by understanding the nature of the traversal, it becomes possible to fully retrieve the contents of the assisting stack from the algebraic tree description of the stack top node. The definition of a function `worklist` doing exactly that is given in Fig. 11. The function returns a list of subtrees (the second component of its return value) in addition to the coordinates of their roots (the first component of its return value). By applying `worklist` on the coordinate of the stack top node, the contents of the assisting stack can be then recovered from the root identifiers of the returned list of subtrees. The list of coordinates will be used later in Sec. 3.3.3.

3.3.3 The Cornerstone Loop Invariant.

At last, we come to present a recipe for stating loop invariants to verify non-recursive traversals using the *cornerstone* invariant, which is defined as an inductive Coq predicate in Fig. 12 and is parameterised by the tree `tr` being traversed. It relates assisting stack (captured in its first index of type `list B`) and the pre-iteration visited prefix (*i.e.*, its second index of type `tree`). In order to use the cornerstone invariant, one has to assume that the root has been visited before the loop begins and that its children nodes are in the assisting stack at the start of the loop. In practice, this is almost always the case, and it is the case for all the operations of tree clocks (*cf.* Sec. 4).

```
Fixpoint worklist tr pos : (list (list nat)) * (list tree) :=
  let 'Node a chn := tr in
  match pos with
  | nil => (nil, nil)
  | x :: pos' =>
    match nth_error chn x with
    | Some ch =>
      let (res1, res2) := worklist ch pos' in
      (map (fun i => i :: nil) (seq 0 x) ++
       map (fun l0 => x :: l0) res1, take x chn ++ res2)
    | None =>
      (map (fun i => i :: nil) (seq 0 (length chn)), chn)
    end
  end.
```

Fig. 11. Retrieving a worklist given a tree `tr` and a node position `pos` in it. The auxiliary function `seq(n, m)` returns a list of natural numbers `[n; n+1; ...; m-1]`.

During a loop iteration where the assisting stack is not empty, according to the `TInv_intermediate` case in Fig. 12, the stack top node (*i.e.*, the root of `sub`) will be popped, and its children nodes will be pushed into the stack before the iteration ends.[4] The lemma in Fig. 12 serves to re-establish the invariant at the end of the iteration.

A notable feature of this pure loop invariant is its *extensibility*: it characterises only the core components of non-recursive traversal, thereby granting users the flexibility to use it in conjunction with other invariants. In particular, one can use this loop invariant as the cornerstone of a “larger” loop invariant specific to a concrete non-recursive traversal. At a high-level, any such larger loop invariant can be constructed using the following *invariant template*:

$$\begin{aligned}\mathsf{loopinv}(tr, p) \triangleq \exists\, stk, pf,\ & \\ & \lceil \mathsf{traversal\_invariant}(tr, stk, pf) \land \cdots \rceil \ast \\ & \mathsf{tree\_rep}_{\mathsf{arr}}(p, tr),\end{aligned}$$

where *tr* is the tree being traversed. We phrase the template using the array view predicate, since the structure of the original tree *tr* is usually not changed during traversal, and the contents of *tr* are retrieved via random accesses.

3.3.4 Traversal Invariant in Action.

We conclude our presentation of Arboreta by revisiting the verification challenge posed by the `copyval_and_move` function from Fig. 4a.

[4] To align with the array-based stack implementation, in this invariant the `list`-based stack has its entry/exit point at the tail position.

```
Inductive traversal_invariant tr : list B -> tree -> Prop :=
  | TInv_terminate : traversal_invariant tr nil tr
  | TInv_intermediate : forall stk pos sub,
    locate tr pos = Some sub ->
    stk = (map id_of (snd (worklist tr pos) ++ (sub :: nil))) ->
    pf = pre_iter_visited_prefix tr pos ->
    traversal_invariant tr stk pf.

Lemma traversal_invariant_trans tr pos sub :
  locate tr pos = Some sub ->
  traversal_invariant tr
    (map id_of (snd (worklist tr pos) ++ chn_of sub))
    (post_iter_visited_prefix tr pos).
```

Fig. 12. An extensible loop invariant for non-recursive traversals. In the `TInv_terminate` case, the assisting stack is empty and the traversal terminates. The `TInv_intermediate` case snapshots the stack content and the pre-iteration visited prefix given the coordinate pos of the stack top node.

The proof of its SL specification with regard to a functional reference implementation f can be completed by instantiating the invariant template as follows:

$$\begin{aligned}&\exists stk, pf,\\ &\lceil \mathsf{traversal\_invariant}(tr_2, stk, pf) \land tr'_1 = f(tr_1, pf) \land \cdots \rceil \ast\\ &\mathsf{tree\_rep}_{\mathsf{arr}}(p_{\mathtt{tree}}, tr'_1) \ast \mathsf{tree\_rep}_{\mathsf{arr}}(p_{\mathtt{tree2}}, tr_2)\end{aligned}$$

4 Verifying the Tree Clock Data Structure

This section showcases the definitions and techniques of Arboreta working in tandem for verifying a large case study: an executable C implementation the tree clock structure [22].

4.1 Tree Clocks: A Primer

Dynamic analysis is a *de facto* preferred method for detecting concurrency bugs such as data races in multi-threaded programs. Dynamic data race detectors such as ThreadSanitizer [33] and FastTrack [12] observe events of an execution during runtime, infer the *happens-before* (HB) partial order [18] between them and report conflicting events unordered by HB to be in a data race. These detectors play a vital role in revealing data races which may lead to critical failures in large software systems and have been extensively applied in industrial settings.

Instead of explicitly constructing the HB partial order as a graph of events, such tools leverage *time-stamping* to implicitly infer the HB partial order. Analyses such as data race detectors maintain *logical clock* data structures to compute the *timestamp* of each event accurately, potentially performing *clock operations* at every event. A timestamp is a mapping from the identifiers of threads to their respective *(local) clocks* (represented by natural numbers), and the HB relation between events can be recovered by comparing their timestamps. *Tree clock* [22], a recently proposed logical clock variant, achieves *optimal* asymptotic complexity in performing clock operations by novelly exploiting the hierarchical structure of tree.

A logical clock can be regarded as an *abstract data type* which exposes its maintained timestamp and usually supports two clock operations: *join* and *copy*. When implemented in an imperative language, the timestamp is typically mutable, and the two operations work by modifying the timestamp stored in one of the two operands (*i.e.*, they are in-place operations). For an instance C of a logical clock, we use $C{\cdot}\mathtt{val}$ to denote the timestamp that C points to. Let C_1 and C_2 be two instances of a logical clock initially pointing to timestamps T_1 and T_2 respectively. The in-place join operation $C_1{\cdot}\mathtt{join}(C_2)$ should update C_1 so that $C_1{\cdot}\mathtt{val} = T_1 \sqcup T_2$, where $\sqcup$ denotes the *logical join* over timestamps:

$$T_1 \sqcup T_2 \quad = \quad \lambda t.\, \max\{T_1(t), T_2(t)\} \qquad (\textsc{LogicalJoin})$$

Likewise, the copy operation $C_1{\cdot}\mathtt{copy}(C_2)$ should update C_1 so that $C_1{\cdot}\mathtt{val} = T_2$ after the operation is performed.

The vector clock [11, 23] is the traditional logical clock data structure; it represents timestamp using an array, indexed by identifiers of threads. Join and copy operations for the vector clock data structure take $\Theta(k)$ time, where k is the number of threads in the execution. In the context of data race detection, for executions with many events, this can result in prohibitively significant slowdowns. On the other hand, the tree clock internally organises timestamp hierarchically as a tree, where nodes in the tree correspond to threads. Join and copy operations on trees are implemented through tree traversals, and tree's hierarchical structure allows for pruning of the traversal, which is the key to its optimal time complexity.

Formally defined, a tree clock is a tuple $TC = (\mathsf{tr}, \mathsf{ThrMap})$, where tr is a tree such that every node n in the tree is a tuple $n = (t, \mathsf{c}, \mathsf{ac}, \mathsf{p}, \mathsf{ch}, \mathsf{rs})$, where thread identifier t, clock c, and *attached clock* ac are scalar fields, while parent p, head child ch, and right sibling rs are pointer fields to other nodes in tr. Every node except tr's root has an attached clock; for tr's root, its attached clock is undefined and thus marked as $\bot$. Intuitively, the thread at the root of tr "owns" the tree clock instance. Node n_1 is a child of n_2 if the root thread was "made aware of" the thread $n_2{\cdot}t$ (using a message/synchronisation operation m) via $n_1{\cdot}t$, and the attached clock $n_2{\cdot}\mathsf{ac}$ is the clock of $n_2{\cdot}t$ when this message m arrived at $n_1{\cdot}t$. Therefore, for each node n, the attached clocks of its children should be no more than $n{\cdot}\mathsf{c}$. Moreover, its children are arranged in decreasing order of their attached clocks, which facilitates pruning during traversal. $\mathsf{ThrMap}(t)$ identifies the unique node n in tr with identifier t and serves as the timestamp. We provide the visual representation of two exemplary tree clock instances TC_1 and TC_2 in Fig. 15.

In this paper, we focus on formalising and verifying the join operation of array-based tree clocks. For efficiency in an intensive application like data race detection, the tree

```
struct Clock { int clock_clk, clock_aclk; };
struct TreeClock{
  int dim, root_tid, top, *S; // S: assisting stack
  struct Clock* clocks;
  struct Node* tree;
};
void join(struct TreeClock* TC2, struct TreeClock* TC1){
  // ...
  while (TC2->top >= 0){
    int v_tid = TC2->S[TC2->top--];
    struct Clock* uprime_clocks = &(TC1->clocks[v_tid]);
    struct Node* u_node = &(TC2->tree[v_tid]);
    struct Clock* u_clocks = &(TC2->clocks[v_tid]);
    int u_clock = u_clocks->clock_clk;
    if (!node_is_null(u_node)){
        detach_from_neighbors(TC2, v_tid, u_node);
    }
    u_clocks->clock_clk = uprime_clocks->clock_clk;
    u_clocks->clock_aclk = uprime_clocks->clock_aclk;
    struct Node* uprime_node = &(TC1->tree[v_tid]);
    int y = uprime_node->node_par;
    push_child(TC2, y, v_tid, u_node);
    get_upd_nodes_join_chn(TC2, TC1, v_tid, u_clock);
  }
}
```

Fig. 13. Fragment of `join` code and related `struct` definitions.

clock data structure is implemented using arrays instead of explicit pointers. In this case, the join of a tree clock TC_1 into TC_2 (TC_2·`join`(TC_1)) is performed by traversing TC_1 and updating the corresponding nodes in TC_2, in which the traversal is loop-based with the assistance of a stack. We show a snippet (in C) of the join operation on array-based tree clocks in Fig. 13.

4.2 Tree Clocks in Coq, Functionally

We start our tree clock mechanisation by developing its reference implementation in Gallina, the pure functional programming language of Coq. We will then use it in Hoare-style specifications to connect the reference implementation with the C code, following the relatively conventional two-layer paradigm for verifying imperative programs [3].

4.2.1 Datatypes. In Fig. 14a, we model the tree part of the tree clock as the generic functional RLT in Sec. 3.1 with the type parameter `A` instantiated to be the following record type and the type parameter `B` instantiated to be `thread`. The `id_of_data` is thereby set to be the `tid` getter of the record. Here `thread` is also a type parameter and will be instantiated as natural numbers in proving the imperative join operation. Note that compared with the axiomatic definition of a node (*i.e.*, defining a node as a tuple), the data held by a `treeclock` node does not contain the pointer fields: it is implicitly captured in the functional RLT structure.

We model ThrMap via the function `find_node` introduced in Sec. 3.1, and the timestamp from a tree clock (*i.e.*, the mapping from the identifier of a thread to its clock value) can be further expressed as getClock funciton in Fig. 14b.

```
Parameter thread : Type.
Record tc_nodedata : Type :=
  mkTCdata { tid : thread; clk : nat; aclk : nat }.
Definition treeclock : Type := tree tc_nodedata.
```

(a) Functional model of the tree part of tree clock.

```
Definition getClock t tc :=
  match find_node t tc with
  | Some res => (data_of res).(clk) | _ => 0 end.
```

(b) Functional model of the timestamp maintained by tree clock.

```
Fixpoint getUpdatedNodesJoin tc tc' : treeclock :=
  let fix aux tc clk chn : list treeclock :=
  match chn with
  | nil => nil
  | tc' :: chn' =>
    let: Node (mkTCdata v' clk_v' aclk_v') chn_v' := tc' in
    if clk_v' <=? (getClock v' tc)
    then (if aclk_v' <=? clk
          then nil else (aux tc clk chn'))
    else (getUpdatedNodesJoin tc tc') :: (aux tc clk chn')
  end in
  let: Node data_u' chn_u' := tc' in
  Node data_u' (aux tc (getClock data_u'.(tid) tc) chn_u').

Fixpoint detachNodes pf tc : treeclock * list treeclock :=
  let: Node data chn := tc in
  let: (chn', res) :=
    List.split (map (detachNodes pf) chn) in
  let: (res', chn'') := List.partition
    (fun tc' => isSome (find_node (id_of tc') pf)) chn' in
  (Node data chn'', (List.concat res) ++ res').

Fixpoint attachNodes forest tc' : treeclock :=
  let: Node data_u' chn' := tc' in
  let: u_pre := find (has_same_id data_u'.(tid)) forest in
  let: chn_u := (match u_pre with Some u => chn_of u
                                  | None => nil end) in
  Node data_u' ((map (attachNodes forest) chn') ++ chn_u).

Definition corejoin (tc pf : treeclock) : treeclock :=
  let: (Node data_z chn_z, forest) := detachNodes pf tc in
  let: Node (mkTCdata w clk_w _) chn_w :=
    attachNodes forest pf in
  Node data_z
    ((Node (mkTCdata w clk_w data_z.(clk)) chn_w) :: chn_z).

Definition join (tc tc' : treeclock) : treeclock :=
  let: mkTCdata z' clk_z' aclk_z' := data_of tc' in
  if clk_z' <=? (getClock z' tc)
  then tc else (corejoin tc (getUpdatedNodesJoin tc tc')).
```

(c) Functional version of the tree clock join operation.

Fig. 14. Tree clock operations defined in Coq.

4.2.2 Operations. In the original tree clock presentation, the join operation builds upon three auxiliary operations: getUpdatedNodesJoin, detachNodes, and attachNodes. All of them are modelled as recursive functions on tree clocks

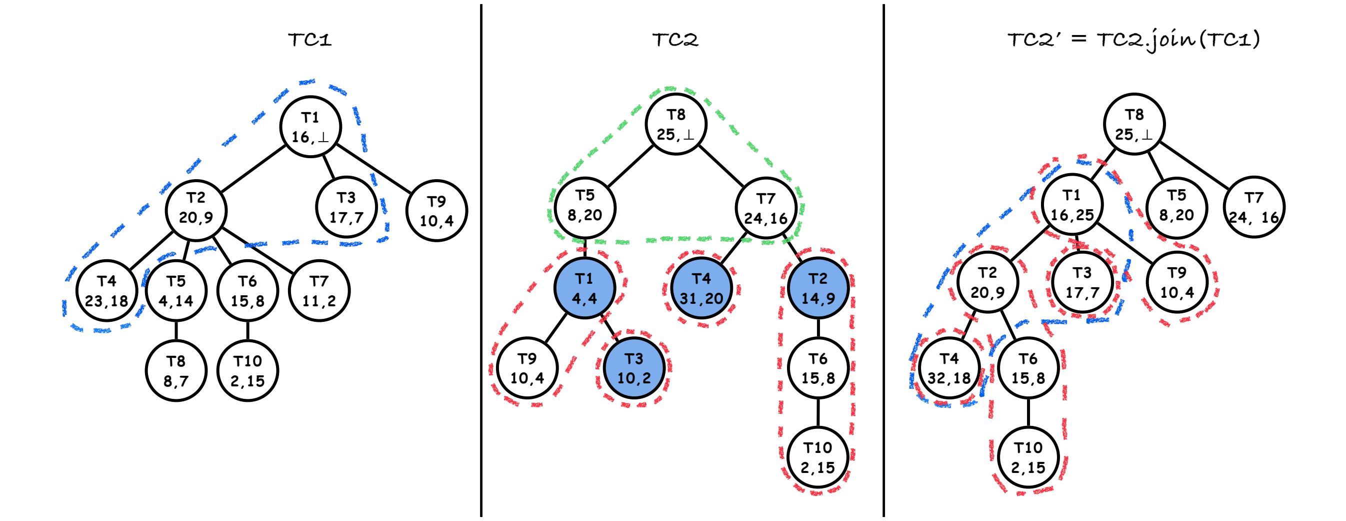


Fig. 15. Illustrative example of joining TC_1 into TC_2. Each node is annotated with its thread identifier, its clock and attached clock (*e.g.*, for node t_2 on TC_1 its clock is 20 and its attached clock is 9). The portion encircled by the blue dashed line is the portion of TC_1 returned by getUpdatedNodesJoin(TC_2, TC_1) (denoted as pf) such that a thread identifier t appears in pf if and only if getClock(TC_2, t) < getClock(TC_1, t). And pf is a prefix of TC_1. detachNodes(pf, TC_2) partitions TC_2 into a "pivot" tree (encircled by the green dashed line) and a list of trees (denoted as *forest*, here containing the trees encircled by the red dashed lines), such that (1) for every tree in *forest*, its root thread identifier appears in pf and no other thread identifier appearing in pf also appears in that tree; (2) the pivot tree is a prefix of TC_2 and does not contain any thread identifier appearing in pf. After that, attachNodes(*forest*, pf) "attaches" all the trees in *forest* to pf by matching the root thread identifiers in *forest*; the resulting tree is the one surrounded by red and blue dashed lines in TC_2'. Finally, the result of join TC_2' is given by making the result of attachNodes be the first child of the root of the pivot tree.

and presented in Fig. 14c. To streamline the proof of the imperative join, we model the core part of join into corejoin, with join, the actual functional join operation, being its wrapper (*cf.* Fig. 14c). While the imperative join TC_2·join(TC_1) modifies TC_2 into another tree clock TC_2', the corresponding functional join, *i.e.* join(TC_2, TC_1), just returns TC_2'. Fig. 15 shows a concrete example of execution of join(TC_2, TC_1).

4.2.3 Predicates and Properties.

We define several extrinsic predicates to ensure well-formedness of functional tree clocks. Specifically, the predicate valid(TC) is a conjunction of NoDupId(TC) and the conditions in Sec. 4.1 that a tree clock should satisfy. We also define the binary predicate respect(TC_1, TC_2) to be the conjunction of the *direct monoticity* and *indirect monoticity*, which are required in the following proofs. Due to space limit, we omit its concrete statement and refer interested readers to its provenance [22].

To guarantee that the functional tree clock implements the interfaces of logical clock correctly, we need to prove that LogicalJoin holds for the functional join operation. With the timestamp model from Fig. 14b, this can be phased as:

$$\begin{aligned}&\forall TC_1, TC_2, \mathsf{valid}(TC_1) \land \mathsf{valid}(TC_2) \land \mathsf{respect}(TC_1, TC_2) \implies \\ &\qquad \forall t, \mathsf{getClock}(t, \mathsf{join}(TC_2, TC_1)) = \\ &\qquad\qquad \max\{\mathsf{getClock}(t, TC_1), \mathsf{getClock}(t, TC_2)\}\end{aligned}$$

The proof of this fact closely depends on the property of getUpdatedNodesJoin(TC_2, TC_1) and the lemma from Fig. 7f.

4.3 Verifying the C Implementation

As the second step of our verification task, we ascribe a Hoare-style specification phrased in terms of functional tree clock manipulations from Sec. 4.2, to the C implementation[5] from Fig. 13 and prove that the specification holds.

4.3.1 The C Implementation.

The imperative tree clock joining (*cf.* Fig. 13) is implemented as a non-recursive tree traversal with its control flow similar that of `copyval_and_move` from Fig. 4a. Notably, the imperative implementation is nothing like its functional reference counterpart from Sec. 4.2. In the functional join, the join is done step-by-step: we first obtain the prefix getUpdatedNodesJoin(TC_2, TC_1) in one step, then accomplish all the subtree detaching in another step, and finally do all the attaching. In the imperative version, however, we will detach and attach a single subtree in an iteration during the non-recursive traversal of that prefix; the join is done only after finishing the traversal.

[5] The C implementation of the tree clock is translated from the (unverified) Java implementation accompanying the original paper [22].

Table 1. Rounded formalisation sizes in lines of Coq code.

	Arboreta	Case Study	Total
Pure	1,300	3,200	4,500
VST	1,400	2,000	3,400
Total	2,700	5,200	7,900

4.3.2 Specification of the Imperative Join.

The specification of the imperative join is given by the following Hoare triple, where p_1, p_2 is the pointer to TC_1, TC_2 respectively:

$$\{\mathsf{tree\_rep}_{\mathsf{arr}}(p_2, TC_2) * \mathsf{tree\_rep}_{\mathsf{arr}}(p_1, TC_1) * \lceil \cdots \rceil\}$$
$$\mathtt{join}(p_2, p_1)$$
$$\{\lambda\_.\mathsf{tree\_rep}_{\mathsf{arr}}(p_2, \mathsf{join}(TC_2, TC_1)) * \mathsf{tree\_rep}_{\mathsf{arr}}(p_1, TC_1)\}$$

where the pure assertion in the precondition is almost the same one as required by TC_1 and TC_2 in Sec. 4.2.3, except for some additional numeric conditions required by VST.

4.3.3 Loop Invariant and Proof Outline.

The most interesting part of the proof is the invariant for `while`-loop at lines 9–23 of Fig. 13. Following the strategy outlined in Sec. 3.3.3, we instantiate the invariant template as follows:

$$\begin{aligned}&\exists stk, pf,\\ &\left\lceil \begin{array}{l}\mathsf{traversal\_invariant}(\mathsf{getUpdatedNodesJoin}(TC_2, TC_1), stk, pf)\\ \land\ TC_2'' = \mathsf{corejoin}(TC_2, pf) \land \cdots\end{array}\right\rceil\\ &* \mathsf{tree\_rep}_{\mathsf{arr}}(p_2, TC_2'') * \mathsf{tree\_rep}_{\mathsf{arr}}(p_1, TC_1)\end{aligned}$$

That is, the traversal_invariant predicate from Fig. 12 is instantiated with getUpdatedNodesJoin(TC_2, TC_1) as an argument. This is because during the imperative traversal, the tree prefix getUpdatedNodesJoin(TC_2, TC_1) is not fully visited, hence the invariant needs to "compensate" for this to match the functional reference implementation.

With the main complexity of the invariant factored out as per the core proof principles of Arboreta, the rest of the proof posed little conceptual challenge. For example, the subprocedure `push_child` is a structure-changing operation, which can be modelled in the same way as has been demonstrated for prepend_child in Sec. 3.2.4. The proof is similar for another subprocedure `detach_from_neighbors` (called at line 16 of Fig. 14c) that removes a child.

The last thing to note is the switch between array view and tree view. When verifying the code in Fig. 14c, we keep using the array view when "stepping through" at the lines 10–14 and deal with the random access. We then switch to the tree view to handle the structure-changing subprocedures. At the line 23, we go back to the array view, since `get_upd_nodes_join_chn` (a function used for traversing children nodes) also performs various random accesses.

Table 2. Evaluation: array-based v. pointer-based tree clocks

1	2	3	4	5	6
Trace len.	num.	Avg. len.	Ptr. TC (s)	Arr. TC (s)	Speedup
(0M, 60M]	35	0.14M	0.22	0.16	1.25×
(60M, 112M]	24	102M	162.27	115.32	1.41×
(112M, 136M]	29	125M	206.57	147.22	1.40×
(136M, 215M]	29	169M	222.36	190.72	1.17×
(215M, 1B]	29	391M	657.23	463.32	1.42×
Total	146	31.41	48.90	36.10	1.35×

4.4 Proof Effort

The quantitative data about our verification efforts is given in Tab. 1. The implementation of Arboreta includes pure facts about array-based trees (Sec. 3.1) and separation logic facts about dual views (Sec. 3.2) and amounts to 2.7 kLOC of Coq. The formalisation of tree clocks includes the functional reference specification (Sec. 4.2) as well as specs and proofs for C code (Sec. 4.3), totalling at 5.2 kLOC of Coq.

The size of the verified C codebase, not included into the statistics in Tab. 1, is around 150 lines of code.

4.5 Evaluation and Benchmarks

To demonstrate the practical relevance of our verified C implementation of array-based tree clocks, we incorporated it inside a HB-based dynamic data race detector.

We evaluate the performance of our array-based implementation over the naïve but easier-to-mechanise pointer-based tree clocks. For a controlled evaluation, our data race detector performs analysis on offline traces logged *a priori*; this ensures that both implementations work on the same trace (for each benchmark program). Conceptually, the race detector maintains a fixed number of tree clocks, processes events one by one, updates them at each event, and checks for data races at every access event. Our benchmark suite is derived from prior work [22] and includes 146 traces from different concurrent C/C++ as well as Java applications. For each trace, we measured the time taken by the race detection analysis, using both the naïve pointer-based, as well as our verified array-based tree clock implementations. Our evaluation was conducted on a 64-bit Red Hat Enterprise Linux 8.4 machine with a single CPU core and 256GB RAM.

In Tab. 2, we summarise the results of our evaluation. The table aggregates the trace logs into 5 groups, dividing the entire set into roughly equal sets based on their lengths (number of events). Column 1 and Column 2 represent the range of trace lengths and the number of traces in each group respectively. Column 4 (resp. column 5) reports the (geometric) mean of the time taken by the analysis that uses the pointer-based (resp. verified array-based) tree clock implementation. Column 6 reports the (geometric) mean of the resulting speedup in each group; the speedup for a given benchmark is measured as the ratio of the time taken by the pointer-based implementation and the array-based implementation. Overall, the array-based implementation offers a 35% speedup, thanks to the efficiency offered by random access in arrays. More importantly, now this fast implementation comes with a formal correctness proof!

5 Related Work

Our work contributes to a large body of research on mechanically verified heap-based data structures and algorithms.

Tree manipulations in Separation Logic. Verifying recursive traversals of heap-based trees in SL (mechanised or not) is considered a standard exercise and is featured in a number of papers and teaching materials [8, 9, 27, 30]. Reasoning about arrays in SL is also a well-studied area, in which many problems can be reduce to reasoning about lists [15].

To the best of our knowledge, relatively few works are concerned with array-based tree representations. Barriére specifies B+ trees in VST by providing a representation predicate that facilitates proofs about traversals via heap induction, but is less convenient to reason about random accesses, as is allowed by the array view in our approach [5].

As described in Sec. 3.2.2, our tree view enables the use of localised reasoning rules such as WandFrameUpdate, which, however, is restricted to scenarios involving the manipulation of only one subtree at a time. Advanced techniques have been proposed to address the more complex cases involving multiple subtrees [7] and we plan to integrate them into Arboreta in the future.

Reasoning about *graphs* in Separation Logic frequently requires defining a representation predicates similar to our tree_rep [24, 34, 36]. Even though such predicates facilitate certain kinds of inductive reasoning [16], they impose additional proof obligations related to non-interference between recursive calls that can be caused by deep intrinsic sharing—such obligations would not be necessary for trees.

Our approach of formulating loop invariants for non-recursive traversals is reminiscent of Charguéraud's specialised rules for inductive reasoning about loops [8, §6], but is tailored for array-based trees and the respective representation predicates. We do not exclude a possibility that such loop invariants can be automatically derived from induction hypotheses of equivalent recursive traversals, and we are planning to investigate this research question in the future.

Reasoning about logical clocks. Tree clocks are a particular instance of *logical clocks* [18]: a family of data structures that are frequently used as a mechanism for reasoning about causality of events in concurrent and distributed systems. Mechanised implementations of a simpler version of logical clocks—*vector clocks* [11, 23]—are featured in several existing efforts on verified algorithms for dynamic data race detection [21, 32, 37]. Verified vector clocks are also an important component of certified implementations of *Conflict-free Replicated Data Types* (CRDTs) [14, 19, 20, 26].

We are not aware of any verified implementations of tree clocks, but we believe it should be possible to use our implementation from Sec. 4 as a verified drop-in replacement for vector clocks in some of those systems. The reason we could not do so immediately is the mismatch between the logical foundations of our proofs and the existing implementations of data race detectors and CRDTs. For example, most of the existing mechanised CRDT implementations [14, 26] are verified in Iris [17], whose proofs are therefore not directly composable with ours in VST. Mansky *et al.*'s verified version of FastTrack algorithm [12, 21] features a C implementation partially verified in VST. Unfortunately, its proof relies on bespoke specifications of logical clock operations, making it difficult to plug in our implementation "as-is". We leave these exercises in proof composition to the future work.

6 Conclusion and Future Work

In this work, we have presented a principled methodology for structuring proofs about manipulations with array-based trees in Separation Logic (SL). We implemented the main components of our approach in a Coq library called Arboreta and showcased them on a large case study, verifying an array-based tree structure used in real-world data race detectors. While our current implementation is tied to the VST framework as a Coq embedding of Separation Logic, we believe, the key ideas of our work can be transferred to other SL embeddings, such as CFML [8], HTT [25], and Iris [17] in a relatively straightforward way. Furthermore, our pure reasoning principles concerning RLTs, such as those delineated in Arboreta-P (Sec. 3.1) and those associated with non-recursive tree traversals (Sec. 3.3), should be adaptable to other theorem provers based on higher-order logic. In the future, we are planning to extend our case studies to other array-based tree structures, such as AVL and B+ trees. We are also planning to integrate our verified implementation of tree clocks into the verified data race detector by Mansky *et al.* [21].

Data Availability

The software artefact with a snapshot of the Coq and C developments accompanying this paper is available online [38]. It contains the source code of Arboreta, the tree clock case study, and the harness to reproduce the experimental results with the data race detector described in Sec. 4.5.

Acknowledgments

We thank the anonymous CPP'24 reviewers for their constructive and insightful comments. We also thank Brigitte Pientka and Sandrine Blazy for their efforts as CPP'24 Programme Co-Chairs. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001, MoE Tier 1 grant T1 251RES2108 "Automated Proof Evolution for Verified Software Systems", and MoE Tier 1 grant "Tree Data Structures for Causal Orderings in Data Race Detection".

References

[1] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS, Vol. 6602)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[2] Andrew W. Appel. 2016. Verified Functional Algorithms. *Software Foundations.* Volume 3 (2016). Available at https://softwarefoundations.cis.upenn.edu/vfa-current/.

[3] Andrew W. Appel. 2022. Coq's vibrant ecosystem for verification engineering (invited talk). In *CPP*. ACM, 2–11. https://doi.org/10.1145/3497775.3503951

[4] Patrick Bahr and Emil Axelsson. 2017. Generalising tree traversals and tree transformations to DAGs: Exploiting sharing without the pain. *Sci. Comput. Program.* 137 (2017), 63–97. https://doi.org/10.1016/j.scico.2016.03.006

[5] Aurèle Barrière. 2018. *VST Verification of B+ Trees with Cursors.* Technical Report. Ecole Normale Supérieure de Rennes.

[6] Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. 2019. Proof Pearl: Magic Wand as Frame. https://doi.org/10.48550/arXiv.1909.08789

[7] Arthur Charguéraud. 2016. Higher-order representation predicates in separation logic. In *CPP*. ACM, 3–14. https://doi.org/10.1145/2854065.2854068

[8] Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998

[9] Arthur Charguéraud. 2021. Separation Logic Foundations. *Software Foundations.* Volume 6 (2021). Available at https://softwarefoundations.cis.upenn.edu/slf-current/.

[10] Olivier Danvy. 2022. Fold-unfold lemmas for reasoning about recursive programs using the Coq proof assistant. *J. Funct. Program.* 32 (2022), e13. https://doi.org/10.1017/S0956796822000107

[11] Colin J Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference.* 55–66.

[12] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI*. ACM, 121–133. https://doi.org/10.1145/1542476.1542490

[13] Jeremy Gibbons. 1991. *Algebras for Tree Algorithms.* Ph. D. Dissertation. University of Oxford.

[14] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434323

[15] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. https://doi.org/10.1145/3591221

[16] Aquinas Hobor and Jules Villard. 2013. The ramifications of sharing in data structures. In *POPL*. ACM, 523–536. https://doi.org/10.1145/2429069.2429131

[17] The Iris Project. 2023. Iris: a Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Coq proof assistant. https://iris-project.org/ Online.

[18] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[19] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *POPL*. ACM, 357–370. https://doi.org/10.1145/2837614.2837622

[20] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying replicated data types with typeclass refinements in Liquid Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 216:1–216:30. https://doi.org/10.1145/3428284

[21] William Mansky, Yuanfeng Peng, Steve Zdancewic, and Joseph Devietti. 2017. Verifying dynamic race detection. In *CPP*. ACM, 151–163. https://doi.org/10.1145/3018610.3018611

[22] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *ASPLOS*. ACM, 710–725. https://doi.org/10.1145/3503222.3507734

[23] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms.* North-Holland, 215–226.

[24] Anshuman Mohan, Wei Xiang Leow, and Aquinas Hobor. 2021. Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms. In *CAV (LNCS, Vol. 12760)*. Springer, 801–826. https://doi.org/10.1007/978-3-030-81688-9_37

[25] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *POPL*. ACM, 261–274. https://doi.org/10.1145/1706299.1706331

[26] Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. 2022. Modular verification of op-based CRDTs in separation logic. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1788–1816. https://doi.org/10.1145/3563351

[27] Peter W. O'Hearn. 2012. A Primer on Separation Logic (and Automatic Program Verification and Analysis). In *Software Safety and Security - Tools for Analysis and Verification.* NATO Science for Peace and Security Series, Vol. 33. IOS Press, 286–318. https://doi.org/10.3233/978-1-61499-028-4-286

[28] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

[29] José Nuno Oliveira. 2023. *Program Design by Calculation.*

[30] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Automating Separation Logic with Trees and Data. In *CAV (LNCS, Vol. 8559)*. Springer, 711–728. https://doi.org/10.1007/978-3-319-08867-9_47

[31] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[32] Caitlin Sadowski, Jaeheon Yi, Kenneth Knowles, and Cormac Flanagan. 2008. Proving correctness of a dynamic atomicity analysis in Coq. In *Workshop on Mechanizing Metatheory*, Vol. 8.

[33] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer – data race detection in practice.. In *Proceedings of the Workshop on Binary Instrumentation and Applications.* http://doi.acm.org/10.1145/1791194.1791203

[34] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964

[35] Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

[36] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying graph-manipulating C programs via localizations within data structures. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 171:1–171:30. https://doi.org/10.1145/3360597

[37] James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. 2018. VerifiedFT: a verified, high-performance precise dynamic race detector. In *PPoPP*. ACM, 354–367. https://doi.org/10.1145/3178487.3178514

[38] Qiyuan Zhao, George Pîrlea, Zhendong Ang, Umang Mathur, and Ilya Sergey. 2023. *Artefact for Article "Rooting for Efficiency: Mechanised Reasoning about Array-Based Trees in Separation Logic".* https://doi.org/10.5281/zenodo.10366484

Received 2023-09-19; accepted 2023-11-25