A Recipe for Modular Verification of Generic Tree Traversals

Laila Elbeheiry
MPI-SWS
Saarland Informatics Campus
Germany
lelbehei@mpi-sws.org

Michael Sammler
Institute of Science and Technology
Austria (ISTA)
Klosterneuburg, Austria
michael.sammler@ista.ac.at

Robbert Krebbers
Radboud University Nijmegen
Nijmegen, The Netherlands
mail@robbertkrebbers.nl

Derek Dreyer
MPI-SWS
Saarland Informatics Campus
Germany
dreyer@mpi-sws.org

Deepak Garg
MPI-SWS
Saarland Informatics Campus
Germany
dg@mpi-sws.org

Abstract

Data structures based on trees and tree traversals are ubiquitous in computer systems. Many low-level programs, including some implementations of critical systems like page tables and the web browser DOM, rely on *generic* tree-traversal functions that traverse tree nodes in a pre-determined order, applying a client-provided operation to each visited node. Developing a general approach to specifying and verifying such traversals is tricky since the client-provided per-node operation can be stateful and may potentially depend on or modify the structure of the tree being traversed.

In this paper, we present a *recipe* for (semi-)automated verification of such generic, stateful tree traversals. Our recipe is (a) *general*: it applies to a range of tree traversals, in particular, pre-, post- and in-order depth-first traversals; (b) *modular*: parts of a traversal's proof can be reused in verifying other similar traversals; (c) *expressive*: using the specification of a tree traversal, we can verify clients that use the traversal in a variety of different ways; and (d) *automatable*: many proof obligations can be discharged automatically.

At the heart of our recipe is a novel use of tree *zippers* to represent a logical abstraction of the tree traversal state, and zipper transitions as an abstraction of traversal steps. We realize our recipe in the RefinedC framework in Rocq, which allows us to verify a number of different tree traversals and their clients written in C.

CCS Concepts: • **Theory of computation** → **Program verification**; **Program specifications**; • **Software and its engineering** → **Software verification**.

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779110

Keywords: Tree traversals, higher-order functions, software verification, separation logic, Rocq

ACM Reference Format:
Laila Elbeheiry, Michael Sammler, Robbert Krebbers, Derek Dreyer, and Deepak Garg. 2026. A Recipe for Modular Verification of Generic Tree Traversals. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779031.3779110

1 Introduction

Trees are one of the most ubiquitous data structures in computer systems. Compilers leverage abstract syntax trees (ASTs) to represent the code structure; web browser engines use Document Object Model (DOM) trees to represent the hierarchical structure of web pages; database systems use B-trees for efficient indexing and organization of data; hypervisors manage guest memory spaces using tree data structures known as page tables.

In these applications, tree data structures are typically processed using *tree traversals*. Such tree traversals are often straightforward in the sense that they traverse the tree according to a standard (*e.g.*, depth-first) traversal algorithm, but they are *generic* in the sense that they are *parameterized* over an operation to be performed on nodes of the tree before, after, and/or between recursive traversals of their children. Furthermore, the per-node operations over which they are parameterized may be *stateful*, modifying not only the contents of nodes but the structure of the tree itself.

In this paper, we explore the problem of how to modularly specify and verify generic depth-first tree traversals, in such a way that they can be instantiated with a range of different stateful per-node (or per-subtree) operations.

Prior work has explored the closely related problem [2, 3, 7, 18, 22] of abstracting a tree (or any iterable data structure) into a *sequence* of elements, corresponding to the sequence in which the elements have been traversed, and then to verify specifications that reason about how this sequence is

```c
struct tree { void *val; struct tree *left, *right;};
typedef struct tree *tree_t;

void trav_post_struct(void f(void*, void*),
                      tree_t *p, void *arg) {
  tree_t tptr = *p;
  if (tptr != NULL) {
    trav_post_struct(f, &(tptr->left), arg);
    trav_post_struct(f, &(tptr->right), arg);
    f(p, arg);
  }
}
```

Figure 1. A higher-order depth-first tree traversal.

changed by the traversal. In essence, they view the tree traversal as a kind of *iterator*, which can then be verified against an iterator-style specification. This iterator view is sufficient for many tree traversals, but does not account for traversals that modify the structure of the tree or that perform per-node operations that depend on the tree structure.

Motivating example. As a concrete example of the kind of traversal we wish to be able to specify and verify, consider the traversal `trav_post_struct` (written in C) that is shown in Figure 1. To a first approximation, this is a simple post-order traversal, albeit one that is parameterized by the operation `f` that should be applied at each node. Note, however, that `f` takes a pointer not just to the element stored in the node (`p->val`) but rather to the *entire subtree* `p` being processed. The operation `f` could thus potentially read and mutate not only the contents of the node being processed (as well as `f`'s own auxiliary state), but also the very structure of the tree being traversed. Indeed, this type of functionality is exhibited by traversals in real-world systems, such as the page table walker in Google's pKVM hypervisor[1] and Blitz's Rust-based web renderer.[2] (We will see an example of such a client of `trav_post_struct` in §3.) It is not clear how one could model the behavior of such traversals using the existing sequence-based specification, since the behavior depends on the tree structure, whereas in a sequence-based specification the information about tree structure is abstracted away.

A verification recipe based on zippers. In this paper, we propose a novel method for specifying and verifying generic tree traversals. Unlike prior work, our approach supports traversals that may depend on or mutate the tree structure. To achieve this, we rely on a combination of *higher-order separation logic* and *zippers*.

First of all, to account for the possibility that the per-node operation over which a traversal is parameterized performs unknown stateful operations, we rely on higher-order separation logic. Formally, we develop our higher-order separation logic specs within the RefinedC framework [24]—an automated yet foundational refinement type system for C based on Iris [12–14]—but conceptually our use of higher-order separation logic is fairly standard, following prior work such as Krishnaswami [15].

More interestingly, instead of specifying traversals using the sequence of traversed elements, we specify them using Huet's zipper [11]. The zipper is a natural, convenient data structure for representing the *traversal state—i.e.*, the intermediate state of the tree at any given point during the traversal, together with information about how far the traversal has progressed. With this traversal state in hand, we can naturally express the specification of the traversal in terms of preservation of a *traversal invariant*: a separation-logic predicate that holds on the state of the tree combined with the traversal operation's auxiliary state. Concretely, the specification of the traversal assumes that the traversal invariant holds of the initial tree state and that it is preserved (by the given per-node operation) as the traversal state progresses from one node to the next in a depth-first order. Under this assumption, the traversal specification guarantees that the traversal invariant will also hold of the final traversal state.

We show how to formalize and verify our zipper-based traversal specifications using a multi-step *recipe*. In essence, the recipe provides a logical template, with a variety of parameters that must be instantiated in a systematic way in order to specify and verify a generic traversal implementation, as well as to verify a particular client of the traversal (*i.e.*, a particular choice of the per-node operation). Although the idea of utilizing zippers in traversals is not new [5, 17], our recipe provides (to our knowledge) the first formal account of tree traversal verification in which zippers are employed in the *specifications* of the traversals themselves.

Our recipe is (a) *general*: it applies to a range of tree traversals, in particular, pre-, post- and in-order depth-first traversals; (b) *modular*: parts of a traversal's proof can be reused in verifying other similar traversals; (c) *expressive*: using the specification of a tree traversal, we can verify clients that use the traversal in a variety of different ways; and (d) *automatable*: many proof obligations can be discharged automatically.

To evaluate the effectiveness of our recipe, we integrate it into the RefinedC framework for C verification [24], and use it to specify and verify C-based generic traversals over pointer-based binary trees, array-based binary trees, and variadic arity trees,[3] as well as a range of different clients of these traversals. Despite our focus on RefinedC, we believe our recipe can be used in conjunction with other languages and separation logic-based verification tools as well.

[1] https://android-kvm.googlesource.com/linux/+/998ccc327b14c03861247540ff6f8135a5283621/arch/arm64/kvm/hyp/pgtable.c#268

[2] https://github.com/DioxusLabs/blitz/blob/b8b88a5dabaac26efdad228d3db692521c0acc48/packages/blitz-dom/src/traversal.rs#L124

[3] A variadic arity tree differs from an n-ary tree in that each node could have a different number of children.

Contributions and outline. We introduce a novel recipe for specifying and (semi-)automatically verifying stateful, generic (*i.e.*, parameterized by a per-node operation), depth-first tree traversals that are commonly found in system applications. Concretely:

- In §2, we present our recipe step-by-step, using some simple motivating examples.
- In §3, we showcase the flexibility of our recipe by demonstrating how it supports a number of more complex tree traversals and tree structures such as array-based binary trees and variadic arity trees.
- In §4, we show how we support automated verification of our traversal specifications in RefinedC.

In §5 we discuss related work, in §6 we discuss limitations and future work, and in §7 we conclude. The Rocq mechanization of our results can be found in our artifact [4].

2 Recipe Overview

This section outlines our step-by-step recipe for reasoning about tree traversals. After introducing our running example, we describe the steps needed to define the primitives that constitute specification (§2.1-§2.4), present a formal specification blueprint for traversals (§2.5), and show how to instantiate the traversal invariant to verify different clients (§2.6).

Running example

For ease of illustration, in this section, we use the `trav_pre` traversal from Figure 2 and the `tree_inc` and `tree_is_sorted` clients to introduce the recipe. In §3, we show how the recipe scales to more complex traversals and clients as well as traversals on different tree-based data structures.

Figure 2a shows a pointer-based implementation of binary trees where leaves are represented as `NULL` pointers and internal nodes are represented as pointers to a `struct tree`.[4] The function `trav_pre` implements a higher-order traversal over this tree: it first checks that a leaf has not been yet reached, then it applies the operation `f` on the `val` field of the node and recurses on the children. The first client, `tree_inc`, increments every node in the tree by instantiating `trav_pre` with the operation `inc_f` that increments the `val` pointee, while the second client, `tree_sorted`, uses `trav_pre` to check whether the pre-order listing of the tree elements is sorted. It uses an argument `d` of type `struct map_data` to keep track of the most-recently-traversed element of the tree and whether the elements traversed so far are sorted. To achieve its goal, `tree_sorted` instantiates `trav_pre` with the operation `is_sorted_f` which, when called on a state `d` and node `v`, updates the `last_seen` field in the state as well as the `sorted` flag (lines 5-8 in Figure 2c). Our goal is to provide a specification for `trav_pre` that enables us to verify these two clients.

[4] We instantiate trees with integers for clarity; the full formalization uses polymorphic trees, implemented in our supplementary material.

```
1 struct tree { int *val; struct tree *left, *right;};
2 typedef struct tree *tree_t;
3
4 void trav_pre(void f(int*, void*), tree_t *p, void *arg) {
5   tree_t tptr = *p;
6   if (tptr != NULL) {
7     f(t->val, arg);
8     trav_pre(f, &tptr->left, arg);
9     trav_pre(f, &tptr->right, arg);
10  }
11 }
```

(a) Traversal implementation in C.

```
1 void inc_f(int *val, void *arg) {
2   *val = *val + 1;
3 }
4
5 void tree_inc(tree_t *p) {
6   trav_pre(inc_f, p, NULL);
7 }
```

(b) A client that uses the traversal to increment the tree nodes.

```
1 struct map_data { int last_seen; bool sorted; };
2
3 void is_sorted_f(int *value, void *arg) {
4   struct map_data *d = (struct map_data *) arg;
5   if (*value < d->last_seen) {
6     d->sorted = false;
7   }
8   d->last_seen = (*value);
9 }
10
11 bool tree_is_sorted(tree_t *p) {
12   struct map_data d = { .last_seen = INT_MIN,
13                         .sorted = true };
14   trav_pre(is_sorted_f, p, &d);
15   return d.sorted;
16 }
```

(c) A client that uses the traversal to check whether the elements in the tree are sorted in a pre-order fashion.

Figure 2. A simple traversal and two clients.

A note on notation: throughout the paper, we use Rocq-like syntax vs. mathematical notation to distinguish between the formalisms that were done in Rocq vs. RefinedC in our development. We choose mathematical notation for the latter since a detailed explanation of RefinedC annotations is beyond the scope of this paper.

2.1 Step 1: Defining the Representation Predicate

The first step of our recipe is to define a *representation predicate* that relates the state of the C data structure to a mathematical model. Using such representation predicates is standard in separation-logic-based verification [20, 23]. We define

the tree model and representation predicate for our running example as follows:[5]

```
(* tree is polymorphic in our Rocq development *)
Inductive tree :=
  | Leaf
  | Node (tl : tree) (a : Z) (tr : tree).

is_tree p Leaf ≜ ⌜ p = NULL ⌝
is_tree p (Node tl a tr) ≜ ∃ pl pr,
  p ↦ {a; pl; pr} ∗ is_tree pl tl ∗ is_tree pr tr
```

An additional benefit that we get from this approach is that the specification becomes modular in the tree implementation. We show in §3 how verifying a traversal over array-based trees, that are refined by the same logical tree model as pointer-based trees, boils down to defining a suitable representation predicate—the formal specification that relies on the logical model of the tree does not change.

Invariant-based specification. Given their iterative nature, the behavior of traversals can be specified using an invariant: for any predicate `R` that holds on an abstract state refining the concrete initial program state and is preserved by the traversal (*i.e.*, is preserved by a transition function that we define on the abstract traversal states), `R` is guaranteed to hold on the traversal state that refines the concrete final state:

```
{ is_tree p t ∗ R holds on the initial traversal state ∗ R
                preserved along progression }
                trav_pre(f, p, arg)
{ ∃ t', is_tree p t' ∗ R holds on the final traversal state }
```

Specifying the behavior of traversals using invariants over the traversal state is a common approach (*e.g.*, [2, 7, 22, 26]) analogous to how the behavior of loops is specified using *loop invariants*: assertions that hold on the program state after each loop iteration. The next steps in the recipe describe how to define *traversal states* and *traversal progression*.

2.2 Step 2: Defining the Traversal State

To illustrate the requirements that the traversal state needs to fulfill, consider the `tree_inc` client in Figure 2b which we assume has the following specification:

```
{ p ↦ tptr ∗ is_tree tptr t }
              tree_inc(p)
{ p ↦ tptr ∗ is_tree tptr (tree_inc_pure t) }
```

where `tree_inc_pure` is defined inductively over the logical tree. To verify `tree_inc`, the verifier needs to instantiate the invariant `R` with a predicate over traversal states such that `trav_pre`'s postcondition—that the predicate holds on the final traversal state—implies that every node in the tree has been incremented. This can be achieved by asserting in the invariant that (a) the nodes that have been traversed are incremented, (b) the nodes that are yet-to-be-traversed do not exceed `INT_MAX - 1`, and (c) the tree structure has not changed.

[5]We use Iris's notation $\ulcorner\phi\urcorner$ to embed a pure Rocq proposition ϕ : `Prop` into separation logic.

From this invariant, we identify at least four aspects that the traversal state ought to capture: (a) the old tree model, (b) which nodes have been traversed, (c) which nodes remain to be traversed, and (d) how to construct the full tree from the current position.

To tackle this challenge, previous work defined traversal states in terms of the sequence of elements enumerated so far and a sequence (or predicate) for the upcoming elements [2, 7, 21, 22]. In this work, we take a different approach. We use Huet's *zipper* data structure to define traversal states [11].

Zippers represent a "pointer" into the tree using a pair of a "focused" subtree paired with its surrounding context. Zippers for binary trees can be defined in Rocq as follows:

```
(* path and zipper are polymorphic in our Rocq development *)
Inductive path :=
  | Top
  | Left (sibling : tree) (parent : path) (data : Z)
  | Right (sibling : tree) (parent : path) (data : Z).

Definition zipper := path ∗ tree.
```

`path` represents, in a bottom-up fashion, the choice (left vs. right) that the traversal made at every step on its route from the root to the focus plus all sibling subtrees that were not on the route of the traversal.

Shadow zippers. Recall how the invariant of the `tree_inc` client needs to state that the traversed nodes are incremented. This is a common pattern in traversal clients: they typically need to instantiate the invariant with an assertion that relates the old and new node values. Stating such relation between the current tree (represented as a zipper) and the original tree (represented as a tree) would unavoidably have to "zoom out" from the focused component in the zipper. To avoid this, we incorporate into the traversal state a second zipper for representing the initial tree. The added zipper shadows the original zipper; they both progress the same way, but the shadow one does not change the values in the nodes along the progression.

The traversal state. We finally have all the components needed to define the traversal state:

```
Inductive state_annot := before | after.
Definition trav_state B := state_annot ∗ zipper ∗ zipper ∗ B.
```

We annotate the state with a `state_annot` to distinguish between the same zipper *before* and *after* traversing the focused subtree. The additional component of type `B` encompasses the values of any additional variables that the traversal uses.

Figure 3 shows how the traversal state evolves in a pre-order traversal of a binary tree. The green arrow points to the subtree that is "focused", and nodes/subtrees labeled in green are already traversed.

Advantages of using zippers. Defining traversal states using zippers (together with the shadow zipper augmentation) offers the following benefits:

1. Zipper-based traversal states encode, in a single structure, all the information that the client invariant might need: which nodes have been traversed? which nodes remain to be traversed? how to construct the original and the current trees (for relational invariants)?
2. Zipper-based traversal states can be used to specify traversals that expose the tree structure to the abstract operation `f`: a capability that sequence-based solutions lack. We show two examples of such traversals in §3 (cf. abstract order and structure-changing traversals). Because the structure of the tree gets exposed to the client operation, the invariant may need to assert a property about the *tree*; a sequence of enumerated elements would not suffice.
3. Zippers, as we will show in the next section, allow us to define traversal progression using the zipper progression primitives, which depend on the structure of the tree but not the traversal implementation. This helps minimize verification efforts; once the user formalizes a traversal, they can reuse most of the specification components that they have defined to formalize any other traversal implementation over the same data structure.

2.3 Step 3: Formalizing Traversal Progression

In order to express the specification of `trav_pre` using an invariant that holds on traversal states after each step of the traversal, we need to accurately define how to step the traversal state to simulate a single step in the concrete traversal.

The key insight here is that `trav_pre` progresses in one of two ways: either it is at a point where it will apply the operation `f`, or it is silently moving up or down the tree (via recursive calls and returns). To match this intuition, we distinguish between these two kinds of progressions: *silent steps* and *action steps*. Silent steps correspond to steps that silently progress to the next node and action steps are precisely those associated with applying the operation `f`.

Concretely, we define the functions `silent_step` and `action_step` to perform a single step on the traversal state. We refer to traversal states where the next step is an action step as *action points*. The signatures of these functions are as follows:

```
silent_step : trav_state B → trav_state B
action_step : Z → B → trav_state B → trav_state B
```

We omit the full definitions of the stepping functions for space limitations, but their definitions are typically a straightforward extension of the navigation primitives for moving up and down the currently focused subtree in a zipper which can be found in the original zipper paper [11]. Both `silent_step` and `action_step` move the zipper such that it points to the next node in the tree, but `action_step` additionally mutates the zipper, inserting the new node value (of type `Z`) and auxiliary state (of type `B`) in the traversal state, before moving to the next node.

The traversal states in Figure 3 are obtained by application(s) of these `silent_step` and `action_step` functions. Note that these functions advance both the actual tree zipper as well as the shadow zipper in the traversal state—for the latter `action_step` has the same effect as `silent_step` since the values in the shadow zipper do not change.

2.4 Steps 4 & 5: Instantiating the Auxiliary State and Defining Action Points

To define traversal progression in terms of the above primitives, we need a predicate, `action_point`, to determine, on each traversal state, which of the two stepping functions should be applied next.

In some cases, this predicate does not just rely on "where we are in the tree" (*i.e.*, the zipper component of the state), it also needs information from the auxiliary state (*i.e.*, the extra component of generic type `B` in the traversal state).

For example, if the traversal aborts early when an error occurs, the auxiliary state needs to encompass this "error flag", and the `action_point` predicate should always evaluate to `None` if this flag is raised. Hence, it would not be possible to define the `action_point` predicate before instantiating the traversal state's type parameter `B` to include the error flag. Therefore, before defining the `action_point` predicate, one should instantiate the type parameter `B` in the traversal state (if needed).

Note that we could also define the traversal state in one shot by immediately instantiating the type parameter `B` when defining `trav_state` in step 2. This, however, would make the traversal state definition less modular (*i.e.*, we might not be able to reuse it when specifying another traversal), so we choose to split it into a "general definition" followed by "concretization" steps.

Action points. Once the auxiliary state has been properly instantiated, we can define `action_points`. For our pre-order traversal, the action points occur when the traversal is about to traverse a new (non-leaf) subtree. This can be defined as:

```
(* action_point for pre-order traversal *)
Definition action_point_pre (s : trav_state B) : option Z :=
  match s with
  | (before, (_, node tl a tr), _,_) => Some a
  | _ => None
  end.
```

If `action_point` evaluates to `Some v` on state `s`, then `s` is at a point where the operation `f` should be applied on the value `v`, in which case then the next state is obtained by applying `action_step` on the state `s` together with the new values that are obtained by calling `f`; otherwise, `action_point` evaluates to `None` and the traversal progresses using a `silent_step`. For instance, calling `action_point_pre` on the first state in Figure 3 evaluates to `Some a`, and so the next state is obtained by calling `action_step a' arg1` s_0 where s_0 is the first state and `a'` is the result of applying the operation `f` on `a`.

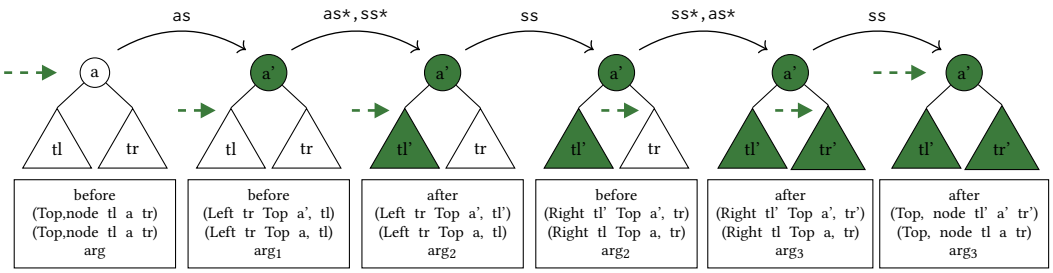


Figure 3. Evolution of traversal states in a pre-order traversal. The steps—denoted with the arrows at the top—are either silent steps (ss) or action steps (as). The shadow zipper encompassing the original tree is omitted from the drawing (but included in the text presentation) for brevity.

```
        ∀ R tptr old_tree old_arg path shadow_z,
    { p ↦ tptr ∗ is_tree tptr old_tree ∗ arg ↦ old_arg ∗
      R (before, (path, old_tree), shadow_z, old_arg) ∗
            □ R_ss_invariant action_point_pre R ∗
          f_preserves_invariant f action_point_pre R }
void trav_pre(void f(int*, void*), tree_t *p, void *arg)
                    { λ _. ∃ new_tree new_arg,
    p ↦ tptr ∗ is_tree tptr new_tree ∗ arg ↦ new_arg ∗
      R (after, (path, new_tree), shadow_z, new_arg) }
```

Figure 4. Specification of `trav_pre`.

2.5 Step 6: Putting It All Together

Now that we have all the ingredients, we set out to write a specification for our pre-order traversal example.

The specification of `trav_pre`, shown in Figure 4, states that, given a predicate `R` over traversal states, if `R` holds on an initial traversal state `s` where the tree `old_tree` is "focused" in `s` (*i.e.*, `s := (before, (p, old_tree), shadow_z, old_arg)`), and `R` is preserved by traversal progression, then `R` is guaranteed to hold on the state after traversing `old_tree`. The assertions `R_ss_invariant` and `f_preserves_invariant` specify that `R` is preserved by the **s**ilent **s**teps and the action steps respectively:

```
Definition R_ss_invariant action_point R :=
  ∀ s, R s -∗ ⌜action_point s = None⌝ -∗ R (silent_step s).

f_preserves_invariant f action_point R ≜
∀ val arg s, {val_p ↦ val ∗ arg_p ↦ arg ∗ R s ∗ ⌜s.2 = arg⌝
        ∗ ⌜ action_point s = Some val ⌝ }
                    f(val_p, arg_p)
{_. ∃ new_val new_arg, val_p ↦ new_val ∗ arg_p ↦ new_arg ∗
       R (action_step new_val new_arg s) }
```

In other words, if `f` is applied at a state where `val` is the next mutant and `arg` is the current value of the abstract argument, then `R` holds after taking an `action_step` on the traversal state using the new element `new_val` and argument `new_arg` that `f` computes.

Note that both `R_ss_invariant` and `f_preserves_invariant` are parameterized by the `action_point` function, which we instantiate using `action_point_pre` in our specification in Figure 4. This modularity allows us to easily reuse the specification blueprint for a number of different traversals.

2.6 Step 7: Verifying Traversal Clients

We can now use this traversal specification to verify the `tree_inc` and `tree_sorted` clients. Note that we are assuming that this recipe is being applied in an automated program verifier setting, so the verification is mostly handled automatically by the verifier. We, nevertheless, show the steps that the programmer needs to take and highlight the proof obligations that arise thereof.

At a high level, to verify the client the user needs to: 1) verify the per-node operation, 2) instantiate the invariant `R`, 3) prove that `R` is indeed a traversal invariant, and 4) show that if `R` holds on the final traversal state then the client's postcondition is satisfied. Let's first consider the `tree_inc` client.

Tree increment client. Figure 2b implements a `tree_inc` function by instantiating `trav_pre`'s per-node operation with the function `inc_f`.

We start by verifying that `inc_f` has the following specification: $\{\texttt{val} \mapsto \texttt{n}\}\ \texttt{inc\_f(val,arg)}\ \{\lambda\ \texttt{\_.\ val} \mapsto \texttt{n + 1}\}$. Presumably, this can be done automatically in state-of-the-art verifiers with support for reasoning about state.

It remains to verify that `tree_inc` satisfies the following specification:

```
{ p ↦ tptr ∗ is_tree tptr t }
          tree_inc(p)
{ p ↦ tptr ∗ is_tree tptr (tree_inc_rocq t) }
```

To verify `tree_inc`, the verifier first needs to reason about its call to `trav_pre`. The standard approach to modularly verify function calls is to exhale the callee's preconditions, and then inhale its postconditions. Exhaling is a standard term in separation logic verifiers that means relinquishing the required resources from the spatial context and proving the pure assertions, and inhaling has the dual meaning [10].

To exhale `trav_pre`'s preconditions, the verifier needs to instantiate the parameters in `trav_pre`'s specification (Figure 4). Some parameters can be automatically instantiated using different heuristics (*e.g.*, pattern-matching on the assertions in the spatial context). The parameter `R` (*i.e.*, the traversal invariant), however, is not one of those inferrable parameters.

Instead, the user needs to define an invariant of this client and manually instantiate `R`. This limitation is not specific to RefinedC; current state-of-the-art verifiers might have limited support for *loop invariant* inference, but inferring invariants that arise from user specifications is, as far as we know, beyond their capabilities.

Intuitively, `tree_inc`'s invariant should state that the nodes that have been traversed are equal to incrementing the corresponding nodes in the original tree, and the nodes that have not been traversed remain unchanged. To define this, we make use of a utility function `forall_rel` which takes a predicate `P` relating tree elements as well as a traversal state, and ensures that the elements that have been traversed are related to the corresponding elements in the original zipper by the relation `P`, and the ones that have not been traversed are identical to their correspondents:

```
Definition R_inc s :=
  forall_rel (λ x shadow_x, x = shadow_x + 1) s
```

In addition to instantiating `R`, the verifier needs hints for other parameters in `trav_pre`'s specification. In particular, the verifier should initialize the path parameter `p` as `Top`, because initially the whole tree is under focus and the context is empty, and the shadow zipper `shadow_z` to match the original zipper `(Top, old_tree)`.

With all parameters finally instantiated, the verifier can continue exhaling `trav_pre`'s preconditions. The first three assertions can be directly relinquished from the verifier's context. We show below the remaining proof obligations:

`R_inc` holds on the initial state: This follows from a trivial lemma stating that `forall_rel` always holds on initial traversal states. RefinedC automatically proves this, but verifiers with more conservative unfolding strategies might need some hints.

`R_inc` is preserved by silent steps: Or, more formally: `R_inc s → action_point_pre s = None → R_inc (silent_step s)`. Again, this fact follows from a lemma about `forall_rel`: that it is always preserved by silent steps. This intuitively holds because the subset of elements that `forall_rel` asserts to be related by `P` does not change after making a silent step. RefinedC could not automatically discharge this goal, but verifiers with SMT-based automation have better chances at automating this step.

`R_inc` is preserved by action steps: The last obligation in `trav_pre`'s specifications asserts that `incr_f` satisfies the `f_preserves_invariant` predicate.

However, recall that our specification for `incr_f` is agnostic to the traversal: $\{\texttt{val} \mapsto \texttt{n}\}\ \texttt{inc\_f(val,arg)}\ \{\lambda\ \_.\ \texttt{val} \mapsto \texttt{n} + 1\}$. This is a common problem when verifying calls to higher-order functions: the closure has a different specification than that expected by the higher-order function. The typical way that program verifiers (including RefinedC) handle this is by showing that the specification of the concrete closure used to instantiate the higher-order function *subsumes* the specification expected by the function. To prove that `inc_f`'s specification subsumes `f_preserves_invariant`, we need to show that if `s` satisfies `R_inc`, and the next mutant in `s` is `n` (*i.e.*, `action_point_pre s = Some n`), then `R_inc` holds on the state `action_step (n + 1) () s` (the second argument is unit because this client does not use any auxiliary state so `arg` has type `unit`). This proof is substantially simplified by choosing to represent traversal states using zippers because the element that gets mutated is exposed at the top-level of the data structure. More precisely, the verifier needs to prove:

```
R_inc (before, (p, node tl n tr), (p, node tl n tr), ()) ⟹
R_inc (before, (Left tr p (n + 1), tl), (Left tr p n, tl), ())
```

By unfolding `R_inc` and `forall_rel`, the proof boils down to showing that the predicate `(λ x shadow_x, x = shadow_x + 1)` holds on `n + 1` and `n`. In RefinedC, this goal was proven automatically without hints.

Once these three facts are established, the verifier finishes reasoning about the call to `trav_pre` by inhaling its postconditions. The next statement in `tree_inc`'s body is `return`, so the verifier must show that `trav_pre`'s postcondition that was just inhaled (that `R` holds on the final traversal state) implies that `tree_inc`'s postcondition is satisfied. This can be proven via a simple induction over the tree data structure. Again, we needed to provide some automation hints for RefinedC to discharge this obligation, but we believe that even SMT-based verifiers would similarly fall short.

Tree sorted client. To verify `tree_is_sorted`, we need to take the same steps that were taken for the tree increment client. We hence only show how to define the invariant for this client.

`tree_is_sorted` checks that the tree is sorted by traversing the tree and checking that each element is greater than or equal to the previous one. It achieves this by using the state argument `arg` to keep track of the most-recently-traversed element of the tree as well as a boolean flag that accumulates the results of the comparisons made so far. The invariant that `tree_is_sorted` maintains thus needs to ensure that the tuple that refines `tree_is_sorted`'s state variable `arg` has the correct value—*i.e.*, it stores the last seen element and the accumulated result of the comparisons.

We define this invariant with a `fold` over the traversed elements of the current zipper using the function `f` as the aggregating function:

```
Definition f v (last_seen, b) := (v, (v <? last_seen) && b).
Definition R_sorted (b, z, shadow_z, arg) :=
  z = shadow_z ∧ arg = fold b z (min_int i32, true) f.
```

Another invariant that `tree_is_sorted` needs to maintain is that the values of the tree are never mutated (recall that the operation that is passed to `trav_pre` can mutate the tree nodes). This is needed to prove the postcondition that the input pointer points to the same tree when the function

returns. `R_sorted` asserts that `z = shadow_z` to ensure that this invariant is preserved.

In these two clients we make use of utility functions `forall_rel` and `fold` defined over zippers. In our experience, most invariants can be defined using a small library of such functions/relations, and so, we augment our development with a library of these definitions and prove a number of facts about them which we found to significantly cut down the effort needed to verify new clients.

Verifying these clients shows that the choice of representing traversal states using zippers simplifies the proof obligations on the client side. Even when the specification given to the client operation is completely agnostic to the fact that `val` comes from a tree node, the verifier was able to prove that the operation preserves the invariant. Moreover, the extra shadow zipper enables clients to easily state any relation between the nodes in the original tree and the current tree.

2.7 Recap

Figure 5 summarizes the recipe presented in this section. Following these steps, users can write a specification of generic tree traversals and verify clients of such traversals. The recipe can be divided into: steps that are dependent on the concrete implementation tree data structure, steps that are dependent on the logical tree model, steps that are dependent on the traversal implementation, and client steps.

This division minimizes verification efforts by maximizing modularity. For example, steps that depend on the logical model of the tree are modular in the concrete data structure implementation (*e.g.*, if one decides to change the implementation of trees to use arrays instead of pointers, only step one (the representation predicate definition) needs to be repeated). Similarly, to change the implementation of the traversal, one only has to redefine the `action_point` (steps 4 and 5) and adjust the specification footprint if needed (step 6).

Note that this step-component dependence assumes that the traversal moves in a depth-first manner, applying a node operation at certain action points. As we show in §3.3, deviating from this assumption can break this dependence division.

3 Instantiations of the Recipe

To evaluate the scalability of our recipe and check that the additional proof effort required after a change to the traversal implementation is indeed as predicted by Figure 5, we modified the pre-order traversal of §2 in several different ways: post-order, in-order and abstract-order traversals (§3.1), a traversal with early abort (§3.2), a traversal that changes the tree's structure (§3.3), a traversal on trees implemented using arrays instead of pointers (§3.4), and variadic trees (§3.5). All the examples presented in this section are available in our Rocq code in the supplementary material.

Step	Description	Dependent on	Section
1	Define the representation predicate that relates the physical layout of the tree with the logical model	Concrete tree implementation	§2.1
2	Define a `trav_state` data type based on the zipper derivation of the abstract tree model.	Logical model of the tree	§2.2
3	Define the *progression primitives*—`silent_step`, `action_step`		§2.3
4	Instantiate the type argument `B` in `trav_state` to represent any additional state that the traversal relies on	Traversal implementation	§2.4
5	Define the function `action_point`		§2.4
6	Write a traversal-state-based specification using Figure 4 as a blueprint.		§2.5
7	Define a suitable invariant `R` that the traversal needs to preserve for a particular client.	Client	§2.6

Figure 5. The steps of the recipe and the component that each step depends on.

The results of our evaluation are summarized in Figure 6. In addition to these numbers, we have implemented and verified in Rocq a library of utility functions that are useful for verifying clients of binary tree traversals, which amount to 284 lines of Rocq code (LoC). The one-time specification blueprint of Figure 4 needed 21 lines of RefinedC annotations.

3.1 Traversals With Different Order of Operations

Post-order and in-order traversals. We modified the pre-order traversal of Figure 2a to a post-order traversal by swapping line 7 and line 9, and similarly to an in-order traversal by swapping line 7 and line 8. Since the tree data structure did not change, we only had to change the definition of the `action_point` in our recipe (step 5 of Figure 5) to match the implementations of the post-order and in-order traversals. The Rocq changes are minimal, totalling 11 LoC in each case.

Abstracting the order of the traversal. We created an *abstract-order traversal* function that determines when to apply the function `f` based on an argument. This function, inspired by a similar function for walking page tables in pKVM and shown in Figure 7, uses the argument `traverse_flags` to invoke the function `f` on the current node at any client-chosen

Traversal	Step Reuse	Lines of Code (LoC)		
		Spec	**Hints**	**Pure**
Pre-order, §2	-	178	81	0
		LoC relative to Pre-order		
Post-order, §3.1	1–4, 6	7	4	0
In-order, §3.1	1–4, 6	7	4	0
Abstract-order, §3.1	1–3	31	4	0
Early abort, §3.2	1–3	7	75	0
Struc.-changing, §3.3	1–2, 4	31	9	0
Array-based, §3.4	2–5	38	65	404
Variadic trees, §3.5	None	136	171	85

Figure 6. Evaluation of the recipe. The *first line* covers the pre-order traversal of §2: it shows the number of lines of Rocq code needed for the specification (column **Spec**), hints to our automation (**Hints**), and manual proofs to discharge pure side conditions (**Pure**). The *remaining lines* cover all other traversals from §3: they show the recipe steps reused and the lines of Rocq code modified in or added to the pre-order traversal.

subset of the following three points: before the recursive calls to the two children, between them and after them.

For this example, we can reuse steps 1–3 from the recipe. Step 4 needs a change to instantiate the traversal's auxiliary state type to include the three flags in `traverse_flags` that `trav_ao` takes as an argument:

```
Let trav_state := trav_state (B * (bool * bool * bool)).
```

The `action_point` function (step 5), omitted here for brevity, is redefined to select the right combination of the pre-, in-, and post-order points based on `traverse_flags`. We also had to adjust the blueprint specification slightly of the traversal (step 6) to match the new type of `trav_ao`. Overall, the specification of this abstract-order traversal amounted to 31 LoC modified or added relative to the pre-order traversal.

***Illustrative client of* `trav_ao`.** We show below an illustrative client of `trav_ao` that adds the size of the subtree rooted at each `node` to a dedicated field, `node->size`. The client calls `trav_ao` with the following function `subtree_f` for the argument `f`, and the `flags` `call_between` and `call_after` set to `true`.

```
void subtree_size_f(int *value, void *arg, unsigned int cf) {
  tree_node_t node = (tree_node_t) value;
  int *ctr = (int *) arg;
  // in-order call
  if (cf == IN) {
    node->size = *ctr;
    *ctr = 0;
  }
  // post-order call
  if (cf == POST) {
    *ctr = node->size + *ctr + 1;
    node->size = *ctr;
  }
}
```

```
struct traverse_flags {
  bool call_before, call_between, call_after;
};

void trav_ao(void f(int*, void*, unsigned int),
    tree_t *p, void *arg, struct traverse_flags *flags) {
  tree_t t = *p;
  if (t != NULL) {
    if (flags->call_before)
      f(t->val, arg, PRE);
    trav_ao(f, &(t->left), arg, flags);
    if (flags->call_between)
      f(t->val, arg, IN);
    trav_ao(f, &(t->right), arg, flags);
    if (flags->call_after)
      f(t->val, arg, POST);
  }
}
```

Figure 7. A traversal that abstracts over the traversal order.

The call to `trav_ao` with `subtree_f` for `f` has the following invariants: (a) After the recursive call on a `node`'s left child has ended, `node->size` equals the size of the subtree rooted at the left child and `*ctr` is 0, and (b) After the (recursive) call on a `node` has ended, `*ctr` and the node's `size` field are both equal to the size of the subtree rooted at that node.

To see why these invariants hold, note that during the call of `trav_ao` on a node, `subtree_f` is invoked twice: once after the call to the node's left subtree and then again after the call to the right subtree. During the first of these calls, `subtree_f` sets `node->size` to the size of the left subtree, and during the second of these calls it sets both `node->size` and `*ctr` to the size of the entire tree rooted at the node.

Thanks to the use of the tree zipper, we are able to specify both these invariants easily in our framework, and verify the client using a total of 52 additional LoC of Rocq and 12 LoC of RefinedC.

3.2 Traversal With Early Abort

Next, we implemented a pre-order traversal `trav_pre_abort` that maintains an "error flag" in its auxiliary state and aborts as soon as the flag is raised. The core logic of the recursive case of this traversal is shown below.

```
// returns true if error flag has been raised
if f(t->val, arg) return true;
if (trav_pre_abort(f, &(t->left), arg)) return true;
if (trav_pre_abort(f, &(t->right), arg)) return true;
```

To verify this traversal, we: (a) instantiated the traversal state (step 4) to include the "error flag"; (b) modified `action_point` (step 5) to take this flag into account: if an error occurred, `action_point` evaluates to `None` else it evaluates to

`action_point_pre`; and (c) modified the specification blueprint (step 6) to account for `trav_pre_abort`'s boolean return value.

The bulk of the proof effort for this example (75 LoC) was in hints for RefinedC's proof automation. We explain this effort further in §4.1.

More efficient sortedness checking. We use the early abort traversal to write a client that checks for a tree's sortedness more efficiently that the `tree_is_sorted` client of Figure 2c. The idea is to trigger the early abort as soon as a violation of the sortedness property is found. The client no longer needs a variable accumulating the result of the comparisons. We get rid of the `sorted` field in line 1 of Figure 2c, and replace line 6 with `return true` (*i.e.*, violation detected).

The traversal invariant is very similar to that of `is_sorted` (§2.6). The main difference is that, now, `last_seen` does not have to track the last seen element if a violation has already been detected (line 6 below):

```
1 Definition f v (last_seen, b) := (v, (v <? last_seen) && b).
2 Definition R_sorted (b, z, shadow_z, (last_seen, err)) :=
3   z = shadow_z ∧
4   let (x, b) = fold b z (min_int i32, true) f in
5   err = negb b ∧ ∃ n, last_seen = n ∧
6   if err then ⊤ else n = x.
```

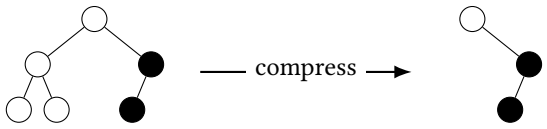
3.3 Traversal With Structure-Changing Operations

Our next example is the potentially structure-changing post-order traversal, `trav_post_struct`, from Figure 1. This traversal's operation `f` takes a pointer to the current node as an argument, allowing `f` to potentially change the entire subtree rooted at the current node (*cf.* other traversals we have discussed so far whose `f` takes only the value stored at the current node as an argument).

The main change needed to verify `trav_post_struct` is in step 3 of our recipe, specifically, in the definition of the function `action_step`, which must now take the new *subtree* rooted at the current node as its first argument and update the zipper to match that subtree (*cf.* other traversals discussed so far wherein `action_step` takes the new *value* stored at the current node and just updates that value in the zipper). This change is tedious but conceptually straightforward. The revised `action_step` function is provided in our supplementary material.

Tree compression client. We used `trav_post_struct` to implement a compression operation that is used in binary *tries* [8]. A binary trie is a tree representation of a finite subset of $\{0,1\}^*$. The left and right children of a node are implicitly labeled `0` and `1`, and a node's value is `true` if the binary string represented by the labels on the (unique) path from the tree's root to that node is in the set being represented, else the node's value is `false`. A natural space optimization is the elimination of subtrees that consist only of `false`-labeled nodes. We call this operation `compress`.

For example, the tree on the left below represents the set $\{\mathbf{1}, \mathbf{10}\}$ (`true`-labeled nodes are black and the rest are white) but it has three redundant nodes. The function `compress` removes these redundant nodes to yield the equally informative but smaller tree on the right.



To implement `compress`, we instantiated `trav_post_struct` with an operation `f` that changes a node's left pointer to `NULL` if its left grandchildren are `NULL` and the left child's label is `false` (and similarly for the right pointer). End-to-end, this compresses the tree by eliminating subtrees that contain `false`-labeled nodes only.

The invariant of this client uses the shadow zipper: it asserts that subtrees that have already been traversed are equal to the compressions of their original counterparts, and the yet-to-be-traversed subtrees are unchanged. To verify this client, we needed 30 lines to specify the invariant, 60 lines of automation hints, and 47 lines of (pure) lemmas to reason about tree compression. (Technical note: to simplify reasoning, our implementation of `compress` assumes that the entire tree is stack-allocated.)

3.4 Traversal on a Different Tree Implementation

Next, we evaluate the proof effort needed to adapt our recipe to a different tree implementation. Specifically, we adapt the pre-order traversal of Figure 2a to operate on trees implemented as overlays on arrays, using array indices in place of pointers as in the prior work of Zhao et al. [27]. The tree implementation and the traversal function are shown in Figure 8. The `node` fields `par`, `lft` and `rgt` are the respective indexes of the `node`'s parent, left child and right child in the underlying array.

We apply our recipe to verify this implementation of `trav_pre_array`. The logical tree model does not change so we can reuse our earlier definitions for steps 1–3. In step 4, we define a new representation predicate to relate the array-based implementation to the logical model. We do this in several steps. First, we define the following `Payload` datatype that abstracts a single node, capturing the value stored at the node and the indexes of the node's parent and two children.

```
(* payload's data is polymorphic in our development *)
Record Payload := { a : Z; par : Z; lft : Z; rgt : Z }.
```

Next, we define a predicate that relates the physical tree, *i.e.*, the array of `struct node`s, to a list of payloads. Finally, we define a predicate to relate a list of payloads to our logical trees. We refer the interested reader to the predicate $\texttt{tree\_rep}_{\texttt{arr}}$ in Zhao et al. [27], or our Rocq development for the full definition of these predicates.

We insert additional pre- and postconditions to the specification in Figure 4, largely because the array-based representation does not enjoy the framing properties that the

```
1 struct node { int *val; int par, lft, rgt; };
2 typedef struct node *tree_t;
3
4 void trav_pre_array(void f(int*, void*), tree_t t,
5     void *arg, unsigned int x) {
6   if (x == -1) return;
7   f(t[x].val, arg);
8   trav_pre_array(f, t, arg, t[x].lft);
9   trav_pre_array(f, t, arg, t[x].rgt);
10 }
```

Figure 8. A traversal over array-based labeled binary trees.

pointer-based representation has. For example, when verifying the traversal in Figure 2a, we can frame any assumptions about `tptr->right` around the recursive call to the left subtree on line 8, but this is not possible in the implementation of Figure 8 because the entire tree is accessible to the callee. This is also the primary source of the overhead of pure lemmas (404 LoC) reported in Figure 6.

3.5 Traversal on Variadic Trees

Our last example is a traversal over variadic trees, where the number of children may differ from node to node.

```
struct ntree { int* val; struct ntree **kids; int numKids; };
```

The main challenge in the verification of a traversal over variadic trees is the need for a loop to traverse the children, as in the following post-order traversal function:

```
void trav_post_variadic(void f(int*, void*),
    ntree_t *p, void *arg) {
  ntree_t tptr = *p;
  if (tptr != NULL) {
    for (size_t i = 0; i < tptr->numKids; i++) {
      trav_post_variadic(f, &(tptr->kids[i]), arg);
    }
    f(tptr->val, arg);
  }
}
```

To verify `trav_post_variadic`, we had to apply all the steps of our recipe from scratch, which also makes this example a good test case for the recipe. First, we defined the logical tree model, the traversal state and the progression primitives as natural extensions of the corresponding definitions for binary trees that we presented in earlier sections (steps 1–3). We used variadic-tree zippers as presented by Huet [11]. Next, we defined the representation predicate (step 4) as follows:

```
is_ntree p Leaf ≜ ⌜ p = NULL ⌝
is_ntree p (Node a chldrn) ≜ ∃ arr,
  p ↦ {a; arr} ∗ ∗_{0≤i≤|chldrn|} is_ntree (arr + i) (chldrn[i])
```

After this, steps 5–7 of the recipe were straightforward.

***Verifying* `trav_post_variadic`.** Unlike our previous examples, verifying `trav_post_variadic` required us to provide a loop invariant to handle the loop that traverses a node's children. This invariant's core assertion is that, at iteration `i`, the traversal invariant `R` holds on the traversal state which reflects updates to the first `i` children but not the rest.

We also had to add assertions to the loop invariant to track the ownership of variables that are used inside the loop and after the loop ends.[6] In total, we added 171 lines of proof-related annotations including 14 lines of RefinedC annotations for defining the loop invariant.

4 Automating the Verification of the Traversal

So far, the recipe presented in §2 shows how to formally specify generic tree traversals. This section focuses on how we can automate the verification of traversals against these specifications.

The main challenge for automating the verification is updating the invariant `R` during the verification of the traversal. For action steps, this happens automatically when recursively calling the traversal: The postcondition of the call automatically updates the invariant to the state after the action step. However, handling *silent steps* is more challenging. They do not correspond to a step in the execution or function call and thus there is no obvious guidance for when these silent steps should be performed.

Consider, for example, the case when the verifier is verifying line 9 in Figure 2a. At this point, the traversal has just returned from the recursive call on the left subtree, guaranteeing `R` holds on the state `(after, (Left tr parent a, tl), ...)`. To show that the precondition of the call on the right subtree holds, the verifier needs to prove `R (before, (?path, tr), ...)`. This mismatch between traversal states in the context and conclusion arises because the traversal, intuitively, makes a silent step between the two recursive calls, but the verifier has not updated its ghost state to reflect this progress. In other words, the verifier needs to be supplied with a heuristic that guides the automation on when to make "silent steps" on the traversal state.

Figure 9 shows the heuristic for silent steps that we propose. Intuitively, this verifier should delay making silent steps until it is at a point were it needs to re-establish the invariant (hence, the pattern-matching on the goal in line 1). At that point, if the state in the conclusion is unifiable with the one in the context, then the invariant can be immediately consumed from the context without any ghost state updates (lines 3-5). Otherwise, in case there is a mismatch, we perform a silent step. Concretely, the verifier updates the traversal state in its context by making a silent step and try proving the goal again (lines 7-9). It is critical that progressing the traversal state via silent steps only happens after failing to unify the goal with the current state; otherwise, the verifier could make too many silent steps overshooting the goal state. Note that it is sound to perform these silent steps since `R` is preserved by silent steps, as discussed in §2.6.

[6] Ownership of variables used after the loop must be asserted in the invariant because RefinedC's verifier does not perform frame inference around loops.

```
1 when proving (R s2 ∗ G):
2   find s1 such that R s1 is in context;
3   if (unifiable s1 s2):
4     remove (R s1) from context
5     continue with proving G
6   else
7     remove (R s1) from context
8     add (R (silent_step s1)) to context
9     continue with proving (R s2 ∗ G)
```

Figure 9. Heuristic for automating silent steps.

We have implemented—and proven sound—this heuristic as hints in the RefinedC verifier, and, with the exception of the `trav_pre_abort` example in §3.2 (explained below), the hints are sufficient for automatically verifying all traversals we considered.

4.1 The Case of `trav_pre_abort`

One thing that is common among traversals is that the number of ghost silent steps that the traversal needs to make is always bounded (usually only one step is needed). For a traversal that aborts early, this is no longer the case. If the traversal aborts due to an error, then the number of silent steps taken to transform the current traversal state to the final state is unknown statically ahead of time.

To circumvent this, we needed to add a new hint that, instead of taking a single silent step as in line 7, tries to prove that `s2` can be reached by making 0 or more silent steps on `s1`. Concretely, the new heuristic can be described by inserting the following lines before line 3 in Figure 9:

```
if the error flag in s1 is set to true:
  remove (R s1) from context
  prove (ss_rel* s1 s2):
  continue with proving G

ss_rel s1 s2 ≜ s2 = silent_step s1
```

Here, `ss_rel*` is the reflexive transitive closure of `ss_rel`.

4.2 Verification Using RefinedC

With the hints described in the previous sections, the verification of traversals and clients becomes a standard RefinedC verification. In this section, we briefly describe our experience using RefinedC to verify the traversals and clients.

The main benefit of RefinedC is that its type system automatically discharges the separation logic reasoning (including for the traversal verification after the extensions described in the previous sections). There are two main tasks that require manual effort: First, proving pure side conditions and, second, instantiating existential quantifiers.

Proving pure side conditions. When the RefinedC proof automation discharges the separation logic reasoning, it generates a set of pure side conditions. RefinedC then attempts to prove the sideconditions using a default strategy (based on Rocq's lia tactic). When this default strategy fails (which often happens when custom definitions are involved), the side conditions are provided to the user to interactively prove them in Rocq. In our case studies, a non-trivial fraction of proof effort (for example, 404 LoC for the array-based traversal in Figure 6) went into manually proving such pure goals. In the future, it would be useful to see if one can further minimize this manual proof effort, for example by integrating SMT solvers into Rocq.

Instantiating existentials. As described by Sammler et al. [24], RefinedC instantiates existential quantifiers based on a set of heuristics. However, these heuristics are incomplete and RefinedC can get stuck on instantiating complex existential quantifiers. In these cases—which appeared multiple times during the verification of our case studies—one has to investigate the RefinedC proof and extend RefinedC's heuristics (*e.g.*, with new simplification rules) to enable RefinedC to successfully instantiate the quantifiers. In the future, it would be interesting to investigate if one can find a design of existential quantifiers in RefinedC that ensures that the quantifiers can always be successfully instantiated.

Debugging failed verification. If the verification fails (*e.g.*, for one of the reasons above), RefinedC emits an error message. This error message contains (sometimes more, sometimes less accurate) information about the problematic line of the program and a list of branching choices that the verification took to reach that line. For unsolvable side conditions, the error message provides the side condition. For other failures (*e.g.*, quantifier instantiation failures or missing separation logic ownership), the error show the proof context of the verification and the goal where the verification got stuck. While this error message itself can give useful hints what the problem is, we often found it useful to use RefinedC's ability to interactively step backwards and forwards through the verification leading up to the error to see precisely what steps RefinedC took and why it got stuck.

5 Related Work

Modular reasoning about higher-order iteration. Prior work [2, 3, 7, 18, 22] provides an approach to formally specifying iterations using the sequence of enumerated elements. This approach is limited to traversals where the per-node operation does *not* depend on the tree structure. This limitation is exhibited by either preventing the client operation from having any side-effects on the data structure being traversed [7, 18, 22], or focusing on higher-order lazy iterators where the per-element operation can have a side-effect on a single element and not the entire data structure [2, 3]. In §3.1 and §3.3, we presented two examples—drawn from real-world systems code—in which the specification must expose the structure of the tree to traversal clients.

There are also several mechanically-verified implementations of higher-order iterations in different proof assistants

and verifiers. Lammich and Lochbihler [16] verify a higher-order eager iterator over set data structures in Isabelle/HOL. Milizia [19] verifies common higher-order functions (*e.g.*, `all` and `map`) over slices in Gobra (a verifier for Go). Why3's standard library includes a generic iterator interface called `Cursor`, which has been instantiated and verified for lists and arrays [6]. These implementations all adopt the sequence-based specifications (following Filliâtre and Pereira [7]), and thus inherit the same fundamental limitations.

To the best of our knowledge, this is the first work to provide a systematic recipe for verifying eager higher-order tree traversals with stateful operations. Beyond supporting stateful operations, our recipe uniquely enables significant reuse of verification effort across different traversals, clients, and tree implementations—a dimension largely unexplored in prior work.

Using zippers. Filliâtre [5] observed that zippers provide a systematic approach for implementing *lazy* iterators over binary trees. As a proof of concept, they implemented pre-order, in-order, and post-order iterators using zippers. As in our work, the rationale for using zippers lies in the fact that the small-step semantics of tree traversals can be captured by a sequence of applications of zipper navigation functions on a zipper that is initially pointing to the root of the tree. Our work, however, focuses on verifying higher-order, stateful, eager traversals implemented in C or other low-level programming languages; whereas he focuses on implementing, in a functional programming language, lazy iterators that are first-order and stateless.

Lorenzen et al. [17] use zippers to implement functional versions of insertion algorithms that restructure binary search trees. The authors verify their implementations in the Iris separation logic framework (the same logic that underlies the RefinedC verifier). Their focus on verifying algorithms that change the tree structure aligns with our goal of supporting tree traversals with potentially-structure-changing operations. However, their work deals with verifying tree insertion implementations (rather than generic traversals) for bespoke tree data structures, namely, move-to-root, splay, and zip trees. Their specifications are thus too complex to generalize to the class of traversals that our work targets. Additionally, in their work, they use zippers to achieve tail recursion: the zipper argument serves to accumulate the context of unvisited subtrees. This differs fundamentally from how we use zippers to formalize the traversal state in a way that exposes all relevant information to the traversal clients.

Verifying tree algorithms using separation logic. A substantial body of work has focused on verifying tree-based algorithms using separation logic [1, 9, 17, 18, 25, 27]. This includes verifying a number of algorithms (*e.g.*, `insert`, `remove`, `lookup`, `merge`, and `rotate`) for red-black binary trees [1, 25], intrusive binary trees [9], array-based trees [27], move-to-root, splay and zip trees [17]. Our recipe, on the other hand, focuses on generic tree traversals that are simpler algorithmically but highly configurable—as demonstrated by the variation of case studies presented. This flexibility necessitates that the specifications we provide are *generic and configurable* and, at the same time, adequately *expressive* to allow modular verification of a range of traversal clients.

6 Limitations and Future Work

Breadth-first traversals. Our work focuses exclusively on depth-first traversals. Extending the recipe to breadth-first traversals presents an interesting but non-trivial challenge. The key difficulty is that traversal states for breadth-first traversals cannot naturally be represented using zippers, which are designed to expose a single focused position and a context of unvisited nodes *arranged hierarchically*. Breadth-first traversals, by contrast, maintain a queue of nodes to visit, making the notion of "focused position" less natural. Defining action points and silent steps for this setting would be significantly more involved, which would likely hurt the automation and render the resulting proofs less reusable. We could instead consider alternative representations for the traversal state for breadth-first traversals; however, this might come at the cost of losing the advantages that zippers provided, *e.g.*, the fine-grained access to the tree structure that zippers provided.

Application to real-world traversals. In this paper, we have verified several case studies inspired by real-world systems code. A next step is to apply the recipe to traversals in practice. The pKVM page table walker is a compelling verification target, seeing as our recipe can handle several of its key traversal features, including (a) early termination, (b) user-defined traversal orders, (c) structure-changing operations, (d) simultaneous use of loops and recursion, and (e) range-delimited traversals. (The last two features were not explicitly studied in this paper, but we believe they are perfectly feasible within our recipe framework.)

Applying the recipe to pKVM presents two primary challenges: defining the representation predicate that abstracts the Arm page table architecture as mathematical trees, and automating the verification. More broadly, verifying production systems such as pKVM is challenging because, in addition to the verification effort itself, we need to ensure that RefinedC (or any verifier of choice) can handle all features of the code (*e.g.*, pKVM enums that are currently not supported by RefinedC) and that it is scalable enough (*e.g.*, the page table representation predicate includes non-trivial bitvector arithmetic that severely degrades performance).

Abstracting over the constructors. Our recipe does not try to exhaustively address all features that can be found in tree traversals in real-world systems. For example, one feature that we do not address in this work is abstracting over the constructors of the tree data structure. This is quite

common (*e.g.*, compiler developers utilize this feature to modularly extend the AST with new constructors without changing the traversal).

7 Conclusion

We presented in this paper a novel recipe for specifying and verifying generic, stateful, depth-first tree traversals with features that are commonly found in real-world systems. The recipe decomposes the verification task into a number of steps, designed in a way that enables both modular reasoning and reuse of verification effort across different traversals and clients. By leveraging zipper-based traversal states rather than flattened sequences, we enable clients to express properties that depend on the tree structure. We instantiated the recipe within the RefinedC verifier and demonstrated its effectiveness on a range of case studies, including traversals with different orders (pre-, post-, in-, abstract-order), early termination, structure-changing operations, and diverse tree implementations (pointer-based, array-based, variadic-arity).

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions. This research is supported in part by generous awards from Android Security's ASPIRE program and from Google Research. The third author is supported, in part, by ERC grant COCONUT (grant no. 101171349), funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Lukas Armborst and Marieke Huisman. 2021. Permission-Based Verification of Red-Black Trees and Their Merging. In *FormaliSE@ICSE*. 111–123. doi:10.1109/FORMALISE52586.2021.00017

[2] Aurel Bílý, Jonas Hansen, Peter Müller, and Alexander J. Summers. 2023. Compositional Reasoning about Advanced Iterator Patterns in Rust. In *IWACO 2023*.

[3] Xavier Denis and Jacques-Henri Jourdan. 2023. Specifying and Verifying Higher-order Rust Iterators. In *TACAS (2) (LNCS, Vol. 13994)*. 93–110. doi:10.1007/978-3-031-30820-8_9

[4] Laila Elbeheiry, Michael Sammler, Robbert Krebbers, Derek Dreyer, and Deepak Garg. 2025. Artifact for A Recipe for Modular Verification of Generic Tree Traversals. https://doi.org/10.5281/zenodo.17799204

[5] Jean-Christophe Filliâtre. 2006. Backtracking iterators. In *ML*. 55–62. doi:10.1145/1159876.1159885

[6] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

[7] Jean-Christophe Filliâtre and Mário Pereira. 2016. A Modular Way to Reason About Iteration. In *NFM (LNCS, Vol. 9690)*. 322–336. doi:10.1007/978-3-319-40648-0_24

[8] Edward Fredkin. 1960. Trie memory. *CACM* 3, 9 (1960), 490–499. doi:10.1145/367390.367400

[9] Marc Hermes and Robbert Krebbers. 2024. Modular Verification of Intrusive List and Tree Data Structures in Separation Logic. In *ITP (LIPIcs, Vol. 309)*. 19:1–19:18. doi:10.4230/LIPICS.ITP.2024.19

[10] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. 2013. Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions. In *ECOOP (LNCS, Vol. 7920)*. 451–476. doi:10.1007/978-3-642-39038-8_19

[11] Gérard P. Huet. 1997. The Zipper. *JFP* 7, 5 (1997), 549–554. doi:10.1017/S0956796897002864

[12] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. doi:10.1017/S0956796818000151

[13] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. doi:10.1145/2676726.2676980

[14] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. doi:10.1145/3009837.3009855

[15] Neelakantan R. Krishnaswami. 2012. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. Ph. D. Dissertation. Carnegie Mellon University, USA. doi:10.1184/R1/6724235.V1

[16] Peter Lammich and Andreas Lochbihler. 2010. The Isabelle Collections Framework. In *ITP (Lecture Notes in Computer Science, Vol. 6172)*. Springer, 339–354. https://doi.org/10.1007/978-3-642-14052-5_24

[17] Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2024. The Functional Essence of Imperative Binary Search Trees. *PACMPL* 8, PLDI (2024), 518–542. doi:10.1145/3656398

[18] Hannes Mehnert, Filip Sieczkowski, Lars Birkedal, and Peter Sestoft. 2012. Formalized Verification of Snapshotable Trees: Separation and Sharing. In *VSTTE (LNCS, Vol. 7152)*. 179–195. doi:10.1007/978-3-642-27705-4_15

[19] Stefano Milizia. 2022. *Verification of closures for Go programs*. Master's thesis. ETH Zurich, Zürich, Switzerland. https://doi.org/10.1007/978-3-642-14052-5_24

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

[21] Mário José Parreira Pereira. 2018. *Tools and Techniques for the Verification of Modular Stateful Code. (Outils et techniques pour la vérification de programmes impératives modulaires)*. Ph. D. Dissertation. University of Paris-Saclay, France. https://tel.archives-ouvertes.fr/tel-01980343

[22] François Pottier. 2017. Verifying a hash table and its iterators in higher-order separation logic. In *CPP*. 3–16. doi:10.1145/3018610.3018624

[23] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. 55–74. doi:10.1109/LICS.2002.1029817

[24] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. 158–174. doi:10.1145/3453483.3454036

[25] Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif. 2022. Separating Separation Logic - Modular Verification of Red-Black Trees. In *VSTTE (LNCS, Vol. 13800)*. 129–147. doi:10.1007/978-3-031-25803-9_8

[26] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *PACMPL* 5, OOPSLA (2021), 1–29. doi:10.1145/3485522

[27] 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 *CPP*. 45–59. doi:10.1145/3636501.3636944