Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Program Logic for Tree Borrows

Master Thesis

Rudy Peterson

September 19, 2025

Advisors: Johannes Hostert, Prof. Ralf Jung

Department of Computer Science, ETH Zürich

Acknowledgments

I thank Johannes Hostert and Ralf Jung for supervising my thesis. In particular, I thank them for helping me design the logical heap. I also thank them for helping me write this thesis, and for providing me with the LaTeX sources for the Tree Borrows diagrams.

I thank the Iris and Rocq-std++ maintainers for upstreaming my merge requests that resulted from my work, and I thank the Iris community for responding to my questions in the Iris chat room. I also thank them for providing me with the LaTeX sources for the Iris and HeapLang style files.

I thank my friends at ETH Zürich, without whom I would not have had the fortitude and strength to complete this master's program. I thank Aaron, Ahmet, Albert, Andreas, Andrew, Andrij, Anna, Calvin, Caspar, Chris $\times$ 2, Daniel $\times$ 2, Elena, Elias, Federico, Felix, Giorgos, Hairong, Jimmy, Konstantinos, Kwok Wai, Lars, Laurenz, Luca $\times$ 2, Lucas, Maciej, Markus, Martin, Matej, Matteo, Max, Michael, Moussab, Nathan, Nicola, Paul, Philippe, Rebecca, Santos, Seungchan, Sven, Teymour, Thomas, Victor, Yuchen, and Yuejiang.

I thank my great aunt, Judith Hollenweger Haskell, and her adopted Ukranian family, for their kindness, support, and boundless hospitality.

I thank my parents, Natalie and Lee Peterson, and my sibling Sacha Peterson, for their bottomless love and support. I am so grateful for my family, and especially for visiting me during my studies at ETH Zürich, giving me the love and strength to press on.

Finally, I thank my cats, Leo and Pico, for being so dang cute, sweet, and loving. I missed them so much during my studies far away in Zürich. Leo and Pico's antics and cuteness gave me so much strength and joy. I am devastated that I lost my little buddy Pico. Pico, knowing you were around, being playful, cute, and snuggly with your brother Leo gave me so much happiness, and I hope we provided you with a loving home and family.

Abstract

The Rust programming language, via its ownership model and borrow checker, provides strong memory safety guarantees for references and aliasing. Compilers take advantage of the guarantees of the borrow checker in order to justify powerful optimizations. Yet, the borrow checker is unable to reason about unsafe code and raw pointers therein, which may bypass Rust's type system and violate its safety guarantees, complicating compiler optimizations. Tree Borrows is a new formal aliasing model for Rust, which precisely articulates the contract unsafe Rust should abide by. Tree Borrows provides compilers with the justification to perform optimizations.

There are many program logics for Rust, but very few consider the aliasing model. This limitation of these program logics precludes one from safely composing safety and correctness proofs in the program logic with semantics-preserving proofs for the compiler. Thus we present a program logic for Tree Borrows capable of modularly reasoning about *ghost trees* in a higher order and concurrent setting under block-based memory. Our novel ghost tree construction unlocks powerful reasoning principles, including pointwise permission weakening, subtree deletion, and lateral separation. Our extensive *Lilac Tree* library buttresses our program logic, enabling these reasoning principles.

Contents

April is the cruellest month, breeding
Lilacs out of the dead land, mixing
Memory and desire, stirring
Dull roots with spring rain.

— T. S. Eliot, *The Waste Land*

Chapter 1

Introduction

The Rust programming language aims to provide robust memory safety assurances via its *ownership model* and *borrow checker*. These mechanisms prevent a large swath of memory safety errors at compile time, even in higher-order and concurrent programs. Rust's ownership system enforces that *any resource may only have one owner at any time*. Furthermore, Rust's borrow checker enforces *aliasing xor mutability*: either there may be multiple immutable aliases of a reference, or there may be a single mutable reference. These restrictions prevent accesses to *dangling pointers*, *data races* between concurrent threads, and even *iterator invalidation* resulting from mutating a collection while iterating over it.

Compilers would like to use the guarantees of the ownership system and borrow checker to justify powerful optimizations, such as reordering reads and writes, inlining reads, etc. However, *`unsafe` Rust* provides programs with the ability to access *raw pointers*, which programs may exploit to evade the strict laws of the borrow checker, such as by aliasing a mutable reference obtained from a raw pointer. `unsafe` Rust is sometimes necessary, and may be used to implement safety-encapsulated abstractions such as for slices, `Vec`, and `HashMap`. Unfortunately, the presence of `unsafe` code may invalidate compiler optimizations. Furthermore, layers of the Rust compiler already makes choices about optimizing code in the presence of aliasing without a formal dynamic semantics, much less a formal *aliasing model*: a precise paradigm within the dynamic semantics which defines which pointers may be used to access memory. In an aliasing model for Rust, we would still like to enforce the uniqueness of mutable references. For instance, the compiler may choose to replace a load from a reference by inlining the value, but this may be unsound if this reference is actually aliased by some other mutable reference.

Enter Tree Borrows [31], a formal aliasing model for tracking permissions of and relationships between references, consistent with Rust's borrow checker.

Tree Borrows *precisely defines* which accesses to references bring about *undefined behavior* (UB), even for references obtained surreptitiously via `unsafe` Rust code and *raw pointers*. Tree Borrows organizes permissions and reference derivation in a tree, where an access (read or write) to one node (one reference) may alter the permissions in all of the other nodes (references), and Tree Borrows forbids an access if it causes UB. Consequently, adding Tree Borrows to the dynamic semantics of Rust provides compilers more UB to exploit in order to justify powerful optimizations: if a source program already incurs UB, then a compiler should have carte blanche to transform and optimize the source code arbitrarily. Villani et al. [31] introduces Tree Borrows to Miri [11], a framework for detecting UB and aliasing violations at runtime for Rust programs, and provides a mechanized implementation of a core calculus in Rocq and Simuliris [8] featuring compiler-correctness proofs for important aliasing optimizations. Lamentably, this additional UB places a greater burden upon efforts to prove program correctness via *separation logic*.

Separation logic is a powerful framework to reason about programming languages that manipulate memory, even concurrently. Higher-order separation logic frameworks such as Iris [16, 14, 15, 18, 26] allow reasoning about higher-order functions and closures in the presence of mutable state and concurrent execution. Iris instances for Rust [13, 6, 7] capture a *core calculus* exhibiting the salient stateful, higher-order, concurrent, and *aliasing* semantics of `unsafe` Rust. *Adequate* Hoare triples for programs in these logics essentially guarantee that *the given program executes safely without UB*. However, this prior work largely *ignores any notion of an aliasing model*, and thus program proofs from this prior work may not in general be safely composed with aliasing optimizations.

In summary, we have Tree Borrows [31], which defines a Rust aliasing model and justifies compiler optimizations, and we have separation logics for Rust [13, 6, 7], but *we are missing* a separation logic for Rust that considers an expressive and flexible aliasing model such as Tree Borrows.

Integrating an aliasing model such as Tree Borrows into a separation logic is no trivial task, since placing further restrictions upon accesses to references introduces further UB into the dynamic semantics. And the more UB in the dynamic semantics of a programming language, the greater the difficulty separation logics face when attempting to reason about program behavior. As programmers and proof engineers, we want to preclude and avoid UB in our programs.

In order to safely compose foundational separation logic correctness proofs with meaningfully optimizing compilers for `unsafe` Rust programs, we must first resolve a *tension* between the desiderata of programmers and proof engineers who prefer to avoid UB, and compiler engineers that benefit from more UB:

1. By broadening the scope of behaviors considered UB, compilers attain more freedom to transform source programs, unlocking more optimizations while preserving the semantics of the given source program.
2. Additional UB creates more obstacles that impede safety and correctness proofs, since more UB introduces further cases that provers need to address or rule out. In particular, it is not clear is there is a *nice and modular* way to reason about these obstacles.

Tree Borrows [31] addresses item 1, but the prior work does not yet adequately address item 2. In this thesis, we attempt to close the research gap for item 2. We incorporate Tree Borrows into the semantics of a core calculus, and lay the foundations of a modular separation logic capable of assuaging the inherit complexity of Tree Borrows. We mechanize our separation logic in the Rocq proof assistant via Iris in order to achieve unassailable mathematical rigor. We intend this work to be another small step in a larger Rust verification effort towards reasoning about realistic `unsafe` programs and providing guarantees about real-word implementations of Rust's safe abstractions.

1.1 Our Contribution

We make the following contributions:

- We build an extensive library for *Lilac Trees* in Rocq, a variant of Rose Trees enjoying stable insertion and deletion. Our libraries support an induction principle for trees, reasoning about tree accesses, properties of pointwise predicates over and relations between trees, monadic transformations over trees, and operations to merge and partition trees.
- We design a core calculus λ^{TB} featuring a simplified Tree Borrows model and block-based memory. This core calculus features non-deterministically generated reference tags, courtesy of our Lilac Trees.
- We engineer a program logic in Iris for λ^{TB} capable of reasoning about mutable, concurrent, and higher-order programs with aliasing under Tree Borrows. Our program logic uses *ghost trees*, which logically relate to the borrow trees in the physical state. Our logical state enables the following reasoning principles for ghost trees:
  - *Pointwise Permission Weakening*: Our ghost trees may *over-approximate* physical trees with *weaker Tree Borrows permissions*. This enables the reconciliation between ghost trees from different branches in the program.
  - *Subtree Deletion*: Our ghost trees may be *structural prefixes* of the physical trees. This enables logically pruning subtrees from the proof state, curbing proof state explosion from successive retags.

– *Lateral/block-wise Tree Separation*: We may partition ghost trees across the heap blocks. This empowers the logic with the ability to reason about concurrent accesses to different locations within the same block. Furthermore, our system is capable of logically re-combining independently retagged partitions.

Amidst working towards our higher level goals, we produce substantial contributions to the Rocq-std++ and Iris libraries. Our efforts lead to 14 accepted and upstreamed merge requests across Iris repositories, which now feature infrastructure originating from our utility layer.

1.2 Thesis Outline

We structure this thesis as follows:

- **Chapter 2 (Background)**: We summarize some basics of aliasing in Rust and describe Tree Borrows. Furthermore, we provide an outline for how to instantiate Iris with a custom *state interpretation (SI)* using ghost resources for finite maps.
- **Chapter 3 (Tree Library)**: We introduce our Rocq library for Lilac Trees, a custom tree data structure. We cover basic operations, and illustrate the key properties of Lilac Trees.
- **Chapter 4 (Program Logic Interface)**: We detail the syntax and semantics of our core calculus λ^{TB}. Then we establish the primitive Hoare logic and points-to laws, and demonstrate how they achieve the core reasoning principles with example λ^{TB} programs.
- **Chapter 5 (Model)**: We dive into the logical state underlying the primitive laws, points-to laws, and reasoning principles of our program logic. We begin with a relatively basic state interpretation, and progressively enhance our proof machinery and refine the logical heap in order to support increasingly more complex reasoning principles.
- **Chapter 6 (Conclusion):** We briefly recap our contributions, provide an overview of the implementation, summarize limitations of our development, and discuss directions for future work and possible extensions to the logic and possible reasoning principles.

1.3 Conventions and Notation

Many operations, such as tree access, are *monadic*, and in order to properly chain these operations together, we require *monadic bind* $\gg=$. We utilize the $\gg=$ notation in chapter 3, where the explicit monadic notation best illustrates the nature of the tree library. In chapter 4 and chapter 5, we drop this notation where it would obscure the more salient ideas concerning the design of the logic. This is especially true of access, which is a monadic operation in Rocq. We decline to notate $\gg=$ and explicit Some and None constructors for option types, and instead implicitly coerce successful results to Some, and write $\perp$ instead of None when the operation fails. Our convention closely mirrors those adopted by other Iris-related works.

In chapter 3, we employ explicit, Rocq-std++-style notation for lookup of the form t !! ks, but in later sections we use more traditional finite map application $t(ks)$ to avoid syntactic clutter. We utilize the same lookup and insert notation for both trees and finite maps. This is consistent with our Rocq development, where both trees and finite maps use instances of Rocq-std++ typeclasses for insert and lookup, so the notations also match in Rocq.

To denote points-to assertions that own a resource at a single offset within a block, we write $\ell \mapsto x$, and to denote points-tos that hold resources for multiple offsets in a block, we write $\ell \mapsto\ast \mathbf{x}$. Furthermore, we notate a resource for multiple offsets with bold font $\mathbf{x}$, similar to vector notation in Physics. If an individual value repeats across offsets, we write $\vec{x}$. We employ these conventions for both values (lists of values) and trees (logical trees spanning multiple offsets in a block at every node).

In the sections exhibiting (section 4.2.2) and explaining (section 5.2, section 5.3) the weakening relation $x_1 \preceq x_2$, we overload the notation $x_1 \preceq x_2$, which at the base level relates permissions and location states, which we then lift pointwise over nodes, and then we lift pointwise over trees. We further overload this notation for the weakening relation between a ghost tree and a physical tree, as well as between ghost trees. In the more advanced versions of the model that include more refined relations between nodes, as well as predicates on right-hand-side-only nodes (section 5.3, section 5.4), we make the details of these refinements implicit to avoid notational clutter. We use the same $x_1 \preceq x_2$ notation for the weakening relation for both block-spanning ghost trees/groves as well as per-location ghost trees. Again, this serves to reduce notational noise. For precise definitions for weakening, we direct the curious reader to the Rocq implementation.

In chapter 4 and chapter 5, we lift the Tree Borrows state machine transition access from individual location states to tree nodes, and then to entire trees. We perform this lifting for both physical trees and the two flavors of ghost trees. We also use access to denote both the access of a single offset z and a set of offsets X.

We refer to a root-only tree Tree$(x, \emptyset)$ with no children as a *singleton* tree. This reflects the conventions of the Haskell Rose Tree library [4], University of Oxford [30], and Carnegie Mellon University [1],

We use the term *lateral separation* to denote partitioning trees apart into disjoint sets of offsets of a block. Visually, we consider the data along blocks to lie “horizontally” with respect to the “vertical”, “downward” growth of the tree data structure from the root. We intend the term “lateral” to evoke the “horizontal” or “side-to-side” dimension of our trees, aligned with the dimension of the blocks.

When describing references and pointers inhabiting some variable `x`, whether from a Rust or λ^{TB} program, we frequently represent the reference/pointer assigned to the variable with the variable itself. For instance, we often write “the permissions of `x`” rather than “the permissions of the reference/pointer assigned to `x`”. This metonymy conveys the salient meaning and avoids distracting pedantry.

Chapter 2

Background

2.1 Rust and Tree Borrows

We begin by outlining the Rust features relevant to Tree Borrows, and then provide a brief introduction to Tree Borrows itself.

2.1.1 Rust References

Rust is a systems programming language renown for its *ownership semantics*, which ensure that only one part of a program may possess full access to or *own* some resource at a time. In addition to read and write access, full ownership grants the ability to deallocate the memory which harbors that resource. Rust's ownership model enables programmers to statically rule out many classes of memory errors that incur *undefined behavior UB*, such as "use after free" accesses to dangling pointers.

Full ownership is unnecessary for many operations and procedures that need only read or write. For this reason, Rust allows programs to *derive references* or *borrow* a pointer to a resource. Rust supports immutable, shared references `&T`, and mutable, exclusive references `&mut T`. Rust's *borrow checker* statically enforces that there is at most one mutable reference to some resource at any given "time". Since the borrow checker surveys the program at compile time, it approximates the interval of execution for which a reference is active with a "span of code" termed *lifetime*. The Rust compiler determines a reference's lifetime by analyzing the encompassing program's control-flow graph. The borrow checker throws a compilation error when it encounters at least two references (of which one or more is mutable) to the same resource with overlapping lifetimes. The Rust community refers to these regulations as the *aliasing xor mutability doctrine*. The borrow checker may end a reference's lifetime upon the derivation of a new reference with conflicting permissions. Consequently, in safe code, the borrow checker prevents *data* *races*, where multiple pointers access the same location concurrently but not synchronously.

Example 1

```rust
fn plus(x: &mut i32, y: i32) {
    *x += y;
}

fn baz() {
    let mut root: i32 = 0;
    let r = &mut root;
    plus(r, 66);
    *r
}
```

Rust also permits deriving references from existing references, dubbed *reborrowing*. This mechanism allows programs to derive a new, child mutable reference from an existing, parent mutable reference. This may appear to violate the "aliasing xor mutability" doctrine, but the borrow checker allows reborrows. Critically, the borrow checker constrains reborrows by requiring the parent reference to *outlive*[1] the child reborrow. Consider example 1. The borrow checker accepts the mutable reborrow for `x` from `r`, and `x`'s lifetime ends upon the conclusion of `plus`.

However, the borrow checker only rules out improper uses of references in *safe Rust*. In `unsafe` Rust code, one may perform accesses directly on *raw pointers*.

Example 2

```rust
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize)
  -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

In a perfect world, one could blissfully eschew `unsafe` Rust, and programmers need only trust the borrow checker. In reality, the cold hard truth is that `unsafe` Rust is a vital escape hatch when the restrictions of the borrow

[1]"In peace, sons bury their fathers. In war, fathers bury their sons", Herodotus.

checker become too cumbersome or even impossible to satisfy. Many modules in the Rust standard library provide client code with *a safe abstraction* atop `unsafe` implementations, especially for data structures, such as `Vec` and `HashMap`. For instance, consider Rust *slices* `[T]`, dynamically-sized contiguous sequences of data in memory. Also consider `split_at_mut` [28], which we show in example 2, which partitions a mutable reference to a slice into two new mutable references with non-overlapping ranges. It is necessary to resort to `unsafe` in order to split up a reference to a slice of type `&mut [T]` into new references, even for disjoint ranges of the slice. This is because the borrow checker does not possess enough information at compile time to determine that the new references indeed refer to non-overlapping memory. Thus one may cast to and from raw pointers to avoid potentially spurious errors from the borrow checker.

Example 3

```rust
fn write_both(x : &mut i32, y : &mut i32) -> i32 {
    *x = 13;
    *y = 20;
    *x // Optimization idea: return 13
}
fn adversary() -> i32 {
    let mut x = 42;
    let ptr = &mut x as *mut i32;
    let val = unsafe {
        write_both(&mut *ptr, &mut *ptr)
    };
    val
}
```

Compilers would like to take advantage of the assurances of the borrow checker to optimize source code. Consider example 3, from Villani et al. [31]. The "aliasing xor mutability" doctrine should entail that `x` and `y` represent disjoint locations in memory. Therefore, the write to `y` should leave `x`'s location unchanged. The compiler may deduce that replacing the final read `*x` with `13` should be sound.

But raw pointers may alias references from safe code, potentially undermining the assurances of the borrow checker. The function `adversary` from example 3 invokes `write_both`. Recall that the inlining optimization for example 3 relies upon the disjointness of the parameter mutable references. Woefully, an adversarial client such as `adversary` may utilize raw pointers in an `unsafe` block to circumvent the borrow checker, and violate its assurances. In a sane invocation, `write_both` would return `13`. Nevertheless, `adversary` passes the same pointer into `write_both` for each argument, causing both functions to return `20`. This appears to render the inlining transformation unsound.

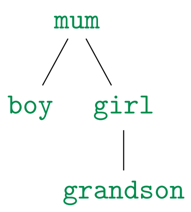
Because Rust's type system does not check memory safety of raw pointers in unsafe blocks, the programmer must manually check that their implementation respects some *aliasing model*. We desire an aliasing model that respects the intuition of programmers and compiler engineers alike. Tree Borrows intends to be this precise, formal contract enforcing proper aliasing decorum.

2.1.2 Tree Borrows

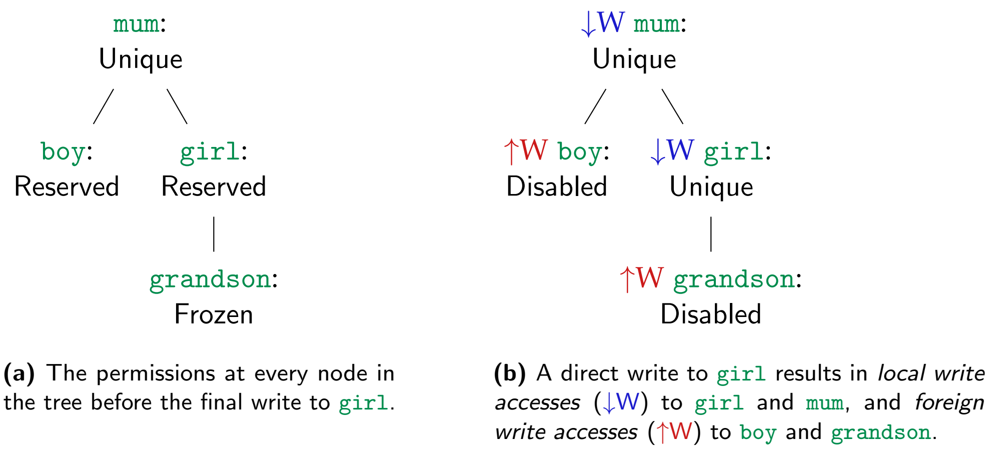
In this section, we provide a crash course introduction for Tree Borrows [31], omitting features out of scope for this thesis. Tree Borrows is a formal aliasing model for Rust, meant to capture the intuition programmers and compiler engineers apply to unsafe Rust. The core conceptual thrust of Tree Borrows relies upon the observation that a natural tree structure emerges from the derivation of references in a Rust program.

Example 4

```rust
fn main () {
  let mut mum = 0;
  let boy = &mut mum;
  let girl = &mut mum;
  let grandson = &*girl;
  *girl = 99;
}
```



Consider example 4. Starting in main, we first allocate memory for mum. We then derive a mutable reference boy from mum, and immediately after we derive another mutable reference girl, also from mum. Then we derive a shared reference grandson from girl, i.e. we immutably reborrow from girl. As depicted in example 4, we may naturally organize the derivation of these references into a tree.



(a) The permissions at every node in the tree before the final write to girl.

(b) A direct write to girl results in *local write accesses* (↓W) to girl and mum, and *foreign write accesses* (↑W) to boy and grandson.

Figure 2.1: Visualization of write accesses to the borrow tree spurred by *girl = 99 from example 4 at the very end of main.

Tree Borrows tracks how accesses to one reference in the tree modify permissions to other references in the tree. For instance, consider the final write to `girl` at the end of example 4. We illustrate the transitions of the Tree Borrows permissions for this access in fig. 2.1. Prior to the final write, `mum`[2] has permission Unique, signifying both read and write permissions. The references `boy` and `girl` both possess permission Reserved, indicating that they each have both read and write permissions, but that they have yet to be activated. The behavior of the Reserved permission reflects the nature of *two-phase borrows*[3] in Rust, which temporarily permit multiple mutable borrows with overlapping lifetimes *before the first write*. As shown in fig. 2.1, the write access to `girl` results in a cascade of changes to the permissions at every node in the borrow tree. Notably, a node's *relative position in the tree* influences how its permission transitions. We distinguish between two complementary types of *access locality*.

- **Local** ↓acc: An access to a node n_{acc} is *local with respect to* a node n_{curr} if n_{curr} is a parent (reflexive-transitively) of n_{acc} in the borrow tree. As shown in fig. 2.1, the access to `girl` is *local* with respect to `mum` since `mum` is a parent of `girl`. Furthermore, `girl` receives a *local* access since `girl` is reflexively a parent of itself.
- **Foreign** ↑acc: An access to a node n_{acc} is *foreign with respect to* a node n_{curr} if n_{curr} is a non-reflexive descendant of n_{acc}, or if n_{curr} is a cousin to n_{acc} in the borrow tree. As shown in fig. 2.1, the access to `girl` is *foreign* with respect to `grandson`, since `grandson` is a child of `girl`. Moreover, the access is also foreign with respect to `boy`, since `boy` is a cousin of `girl`.

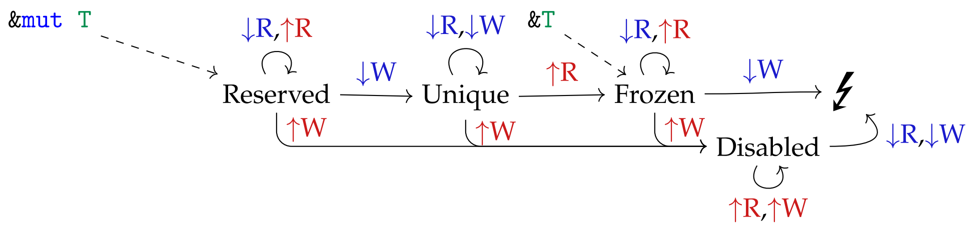


Figure 2.2: Default/unprotected permission state machine diagram from Villani et al. [31], featuring the entry point for mutable references marked by `&mut T`, as well as the entry point for shared references marked by `&T`. Reaching ⚡ indicates that the program has UB. We label transitions by the events that cause them: (R)ead or (W)rite, each either ↑(foreign) or ↓(local).

The *Tree Borrows state machine* for permissions (fig. 2.2 from Villani et al. [31])

[2]Tree Borrows assigns every root node Unique, and since the root only receives local accesses it remains Unique from allocation to deallocation.

[3]The borrow checker only generates a two-phase borrow in special cases, such as for mutable reborrows in fuction arguments. Tree Borrows does not make these distinctions at runtime.

illustrates how each variety of *access events* transitions one permission to another. An access event is a combination of the access locality (local or foreign) and the *access type* (read or write). The root node in a borrow tree always has Unique permission, since the semantics allocates new singleton trees as Unique at the root, and, as shown in fig. 2.2, a Unique node remains Unique under any local access. Since a root node is (reflexive-transitively) a parent to all other nodes in the tree, and Tree Borrows only grows the tree *downward* (corresponding to a reborrow), all access are local with respect to the root. A newly derived shared reference `&T` starts with Frozen permission, which renders local writes UB. In contrast, a newly derived mutable reference `&mut T` begins with Reserved permission, reflecting the reservation phase of two-phase borrows. A mutable reference *activates* and becomes Unique upon receiving its first local write.

Any reference[4] becomes Disabled upon receiving a foreign write. This runtime behavior embodies the "aliasing xor mutability" doctrine of the borrow checker, whereupon the borrow checker ends a mutable reference's lifetime when it encounters a new mutable reference to the same location. Or, in the case of a two-phase reborrow of a mutable reference, the borrow checker invalidates the other references when one is first written to.

We qualify a reference in Tree Borrows by its *position* or *address* in the borrow tree, which we denote as a *tag*. A tag uniquely identifies a reference by providing its corresponding node in the tree. We call deriving a new reference *retagging*, since we are inserting a new node into the borrow tree, which requires a fresh tag. Retags are the runtime analogue to reborrows. Consider again the program and borrow tree in example 4. When the program derives references for `boy` and `girl`, Tree Borrows retags from the root `mum`. Similarly, when the program derives a new reference for `grandson`, Tree Borrows retags from `girl`.

Note that the Tree Borrows behavior for sibling nodes corresponds with the borrow checker's "aliasing xor mutability" doctrine. Upon the first write access to a node in the borrow tree, that directly written-to node becomes Unique, and all of its siblings receive foreign writes, rendering them Disabled. Among its sibling nodes, this newly minted Unique node is "unique" in the sense that it is now the *exclusively accessible reference* among its siblings.

Unlike the borrow checker, Tree Borrows also tracks the permissions of raw pointers. Specifically, Tree Borrows reference tags are included within a pointer's *provenance*, additional pointer state other than the plain memory address. Compilers may exploit Tree Borrows to justify optimizations. We illustrate this with example 3 from Villani et al. [31]. Recall that the compiler

[4]A foreign write *does not* disable a *Reserved interior mutable* node. This thesis does not implement interior mutability, ergo we omit an explanation here. Those interested may consult Villani et al. [31].

may wish to replace the final read access `*x` in `write_both` by inlining `13`. Intuitively, this optimization appears to be sound, since both `x` and `y` are mutable references, which the borrow checker should guarantee concern different memory locations. But also recall function `adversary` from example 3, which demonstrates a means by which an adversarial client may use raw pointers to provide `write_both` with aliasing mutable references. This is where Tree Borrows comes to the rescue. In the original code for `write_both`, Tree Borrows disables[5] permissions to `x` upon the write `*y = 20` when invoked by `adversary`, since `*y = 20` triggers a foreign write to `x`. Consequently, the original program causes UB, which gives the compiler license to perform the optimization.

Thus, compilers exploit the UB brought about by Tree Borrows to justify compelling optimizations. Villani et al. [31] demonstrates the utility of Tree Borrows to compilers performing optimizations via Rocq and Simuliris proofs of semantics-preserving transformations without incurring new UB. Moreover, Tree Borrows was introduced into Miri [11] to test UB in real Rust programs. Tree Borrows incurs less UB than Stacked Borrows (the predecessor to Tree Borrows discussed below), better respecting programmer intuition. Thus Tree Borrows has demonstrated its viability to both Rust programmers and compiler engineers.

Adapting a language's dynamic semantics to include Tree Borrows complicates the language definition. The goal of this thesis is to overcome the challenges of reasoning about programs in the presence of Tree Borrows within the framework of a foundational program logic.

A note about Stacked Borrows Prior to Tree Borrows, researchers developed Stacked Borrows [12], a similar aliasing model for Rust. Additionally, Louwrink [20] developed a program logic for Stacked Borrows. However, Stacked Borrows incurred *too much UB*, triggering UB in commonly accepted `unsafe` programming patterns. For instance, Stacked Borrows does not properly capture accesses to references obtained by *pointer arithmetic* across contiguous memory addresses, whereas Tree Borrows intentionally considers *block-based memory*. In particular, Stacked Borrows forbids accessing pointer offsets outside of the retag range, but Tree Borrows admits such programs. Moreover, Stacked Borrows does not address *two-phase borrows*, which allows the creation of multiple mutable references in a *reservation phase* before their

[5]We elide some details for the purposes of this thesis. Tree Borrows also includes the notion of *protected retags*, which prompt an alternative state machine. The execution of this example under Miri with Tree Borrows triggers UB at `*x = 13`, since `x`'s permission at this point is really *conflicted reserved*. As discussed later on, our work only partially supports protectors, and our core calculus does not include protected retags. The salient point is not *where exactly* UB occurs but that Tree Borrows precludes improperly interleaved writes from different references to the same location.

first write. Lastly, Stacked Borrows is incompatible with some important compiler optimizations, such as read-read reorderings, a deficit which Tree Borrows overcomes. This is a consequence of Stacked Borrows organizing retags into a stack, which is inherently less flexible than a tree. For more details about Stacked Borrows, see Jung et al. [12] and Villani et al. [31]. For a discussion relating Louwrink [20] to our work, see section 6.4.

2.2 Iris

Iris [16, 14, 15, 18, 26] is a concurrent, higher-order separation logic framework capable of reasoning about mutable, concurrent, and functional programs in terms of *resource ownership*. Critically, Iris is highly flexible since it allows the construction of *custom ghost state*, from which one may derive the appropriate separation logic reasoning principles. Researchers have successfully applied Iris to program verification efforts for C [21, 22], OCaml [24, 2, 9], and of course Rust [13, 6, 7]. Program logics such as these instantiate the *program logic layer* of Iris with the *ghost resources* sound with respect to the *physical state* of the dynamic semantics. Specifically, these logics fashion a suitable *state interpretation* (SI), which, in particular for languages with heap semantics, owns an *authoritative* ghost resource reflecting the physical program heap. Here, we outline the fundamentals of ghost resources leveraged by this thesis.

2.2.1 Ghost Maps

Ghost maps are resources that govern ownership of finite, partial maps $K \xrightarrow{\text{fin}} A$. They are described by $\mathit{GhostMap} \triangleq \mathit{Auth}(K \xrightarrow{\text{fin}} (\mathit{DFrac} \times \mathit{Ag}(A)))$, where *DFrac* is the resource algebra of *discardable fractions* [32], $\mathit{Ag}(A)$ is the resource algebra of *agreement* [14], and $\mathit{Auth}(M)$ is the *authoritative monoid* [16].

Consider a finite map $m : K \xrightarrow{\text{fin}} A$. We may construct a ghost map resource from m with an authoritative, fully owned element $\bullet(\mathsf{ghost\_map}(m))$, where $\mathsf{ghost\_map}(m) \triangleq (\lambda x : A.(1, \mathsf{ag}(x)))$ <\$> m, and partially owned fragment elements $k \overset{dq}{\hookrightarrow} x \triangleq \circ([k \leftarrow (dq, \mathsf{ag}(x))])$, where $k \mapsto x \in m$ for some (potentially discarded) fraction dq. When $dq = 1$ in $k \overset{dq}{\hookrightarrow} x$, we may write this as $k \hookrightarrow x$, which entails full ownership of the fragment of m where $k \mapsto x \in m$: we have *exclusive* ownership of $k \hookrightarrow x$. Owning $k \hookrightarrow x$ means that the full 1 trumps the $\mathsf{ag}(x)$ component of $(1, \mathsf{ag}(x))$, allowing one to *update* $k \hookrightarrow x$ to some $k \hookrightarrow y$ by way of GhostMap-Auth-Update if one also owns the authoritative element $\bullet(\mathsf{ghost\_map}(m))$.

When $q < 1$, then the $\mathsf{ag}(x)$ component dominates, meaning if we own $k \overset{q_1}{\hookrightarrow} x_1$ and $k \overset{q_2}{\hookrightarrow} x_2$, then we may conclude that $q_1 + q_2 \leq 1$ and $x_1 = x_2$

GhostMap-Agree
$$k \overset{q_1}{\hookrightarrow} x_1 * k \overset{q_2}{\hookrightarrow} x_2 \vdash q_1 + q_2 \leq 1 \land x_1 = x_2$$

GhostMap-Persistent
$$k \overset{\Box}{\hookrightarrow} x \vdash \Box\, k \overset{\Box}{\hookrightarrow} x$$

GhostMap-Auth-Valid
$$\bullet(\mathsf{ghost\_map}(m)) * k \overset{dq}{\hookrightarrow} x \vdash m(k) = x$$

GhostMap-Auth-Insert
$$\frac{k \notin \mathsf{dom}(m)}{\bullet(\mathsf{ghost\_map}(m)) \vdash |\!\dot{\Rrightarrow} \bullet(\mathsf{ghost\_map}(m[k \leftarrow x])) * k \overset{dq}{\hookrightarrow} x}$$

GhostMap-Auth-Update
$$\bullet(\mathsf{ghost\_map}(m)) * k \hookrightarrow x \vdash |\!\dot{\Rrightarrow} \bullet(\mathsf{ghost\_map}(m[k \leftarrow y])) * k \hookrightarrow y$$

GhostMap-Auth-Insert-Big
$$\frac{m_1 \perp m_2}{\bullet(\mathsf{ghost\_map}(m_1)) \vdash |\!\dot{\Rrightarrow} \bullet(\mathsf{ghost\_map}(m_1 \cup m_2)) * \mathop{\Huge *}_{k_2 \mapsto x_2 \in m_2} k_2 \hookrightarrow x_2}$$

GhostMap-Auth-Insert-Persist-Big
$$\frac{m_1 \perp m_2}{\bullet(\mathsf{ghost\_map}(m_1)) \vdash |\!\dot{\Rrightarrow} \bullet(\mathsf{ghost\_map}(m_1 \cup m_2)) * \mathop{\Huge *}_{k_2 \mapsto x_2 \in m_2} k_2 \overset{\Box}{\hookrightarrow} x_2}$$

Figure 2.3: Iris laws for *GhostMap*.

via GhostMap-Agree. Ownership of $k \overset{\Box}{\hookrightarrow} x$ means that full ownership of $k \mapsto x \in m$ has been *discarded*, and $k \overset{\Box}{\hookrightarrow} x$ is *persistent*, or *freely duplicable* (GhostMap-Persistent).

We may insert multiple new (disjoint) elements into a ghost map via GhostMap-Auth-Insert-Big and GhostMap-Auth-Insert-Persist-Big. Similar lemmas hold for validity and updates (not shown).

If we own $\boxed{\bullet(\mathsf{ghost\_map}(m))}^{\gamma}$ and $\ast_{k \mapsto x \in m} \boxed{k \overset{\Box}{\hookrightarrow} x}^{\gamma}$, then we essentially have an *agreement ghost map*. In an agreement ghost map, we may allocate new persistently owned fragments, but we may never change these fragments.

2.2.2 The State Interpretation and Points-tos

The state interpretation (SI) $S : \mathit{State} \xrightarrow{\mathsf{fin}} \mathit{iProp}$ is a predicate on the physical program state $\sigma : \mathit{State}$ within the *weakest precondition* $\mathsf{wp}\ e\ \{P\}$ connective, which links the ghost program state to the physical program state. We may define the SI for a particular instantiation of irisGS_gen as

$$S(\sigma) \triangleq \exists m. \boxed{\bullet(\mathsf{ghost\_map}(m))}^{\gamma} * m \preceq \sigma$$

where $m : K \overset{\text{fin}}{\rightharpoondown} A$ is some existentially quantified finite map related to the physical state σ by some relation $m \preceq \sigma$. $S(\sigma)$ possesses the authoritative fragment of this ghost map. Typically, σ is or includes a finite map representing the heap. For example, in HeapLang, the sample programming language shipped with Iris, the physical state is essentially[6] a heap $\mathit{State} \triangleq \mathit{Loc} \overset{\text{fin}}{\rightharpoondown} \mathit{Val}$. In general, the domain K of m need not be the domain of σ, and likewise for the image A. In principle, one is free to define S however is most suitable to reason about the ghost resources representing the physical state of the program. The relation $m \preceq \sigma$ enables m to be an *appropriate approximation* of σ for the separation logic. The approximation or weakening in $m \preceq \sigma$ is implementation-specific, but commonly it allows m to represent a subset of σ. Depending on the complexity and layers of *State*, multiple ghost maps m_i, relations $m_i \preceq_i \sigma$, and predicates $S_i(\sigma)$ may constitute the overall state interpretation. We may accordingly summarize the challenge of this thesis as *how to construct a suitable S for a programming language with Tree Borrows*.

The actual ghost resources representing the physical state exposed to users of the program logic typically manifest in form of *points-to* fragments. In HeapLang, the points-to fragment takes the form of $\ell \mapsto_{dq} v \triangleq \boxed{\ell \overset{dq}{\hookrightarrow} v}^{\gamma}$, representing ownership of the heap at location ℓ. In general, points-to connectives possess fragment elements $k \overset{dq}{\hookrightarrow} x$, which represent ownership of some resource x, providing some approximate view of the physical heap.

If we desire a particular ghost resource to be persistent, we may ensure that any element that may have been possibly allocated into the ghost map is persistent in the state interpretation for that resource. We may do this by asserting ownership of every fragment element *from the physical map*:

$$S_{agree}(\sigma) \triangleq \exists m_{agree}. \boxed{\bullet(\mathsf{ghost\_map}(m_{agree}))}^{\gamma_{agree}} * m_{agree} \preceq \sigma * \mathop{\Huge *}_{k \mapsto x \in [\![\sigma]\!]} \boxed{k \overset{\square}{\hookrightarrow} x}^{\gamma_{agree}}$$

where $[\![\sigma]\!] : K \overset{\text{fin}}{\rightharpoondown} A$ is some transformation on the physical state to obtain a ghost map that includes all possible mappings in m, since, as mentioned earlier, $m \preceq \sigma$ (often) entails that m reflects a subset of σ. This method of defining a S_{agree} is a "common trick" in Iris instantiations, such as in Sammler et al. [23].

2.2.3 Primitive Laws

$$\{P\}\, e\, \{v.\, Q\} \triangleq \square\, (P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{v.\, Q\})$$

[6]We neglect mentioning prophecies, since our core calculus does not support this feature.

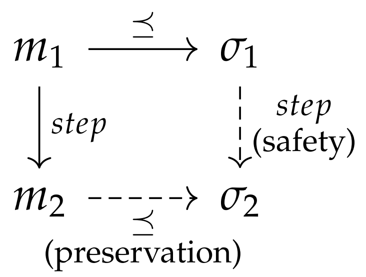
In many Iris instantiations, the *primitive laws* represent the fundamental Hoare logic theorems $\{P\}\, e\, \{Q\}$ invoked by clients of the separation logic. In these logics, the validity of an Iris Hoare triple $\{P\}\, e\, \{Q\}$ entails that for any initial program state σ_1 satisfying the precondition P, any final state σ_2 reachable by the execution of e satisfies Q *and there is no undefined behavior in e's execution*. The precondition P must ensure the conditions that preclude undefined behavior (UB), which includes regulating accesses to references.

WP-LIFT-STEP

$$\frac{\mathsf{expr\_to\_val}(e_1) = \bot}{\begin{array}{l}\forall \sigma_1.\, S(\sigma_1)\ {}_{\top}\!\Rrightarrow\!\!\ast_{\emptyset}\ (\mathsf{red}(e_1, \sigma_1)) \ast \\ \qquad \forall e_2\, \sigma_2.\, (e_1, \sigma_1 \rightarrow_{\mathsf{h}} e_2, \sigma_2) \twoheadrightarrow {}_{\emptyset}\!\!\Rrightarrow_{\top} \Big(S(\sigma_2) \ast \mathsf{wp}^S\, e_2\, \{x.\, Q\}\Big) \\ \vdash \mathsf{wp}^S\, e_1\, \{x.\, Q\}\end{array}}$$

At the heart of an Iris Hoare triple is the *weakest precondition* predicate. When proving $\{P\}\, e_1\, \{v.\, Q\}$ for some particular primitive term e_1, it often suffices to apply WP-LIFT-STEP. When applying WP-LIFT-STEP to a proof goal, one must then essentially prove both of the following:

- **Safety**: Under any state σ_1, there exists some term e_2 and state σ_2 that term e_1 may step to. In other words, the program e_1 should be reducible.
- **Preservation**: Under any states σ_1 and σ_2, and terms e_1 and e_2 such that e_1 under σ_1 steps to e_2 and σ_2, we may re-establish the state interpretation with the new state σ_2.



$$\begin{tikzcd} m_1 \arrow[r, "\preceq"] \arrow[d, "step"] & \sigma_1 \arrow[d, dashed, "\begin{array}{c}step\\ \text{(safety)}\end{array}"] \\ m_2 \arrow[r, dashed, "\preceq"'] & \sigma_2 \end{tikzcd}$$

(preservation)

Figure 2.4: Diagram which demonstrates the cases for weakest precondition proofs. The solid arrows indicate premises, and the dashed arrows indicate the conclusion.

Recall from section 2.2.2 that the SI typically existentially quantifies over some ghost map m related to the physical state σ by some relation $m \preceq \sigma$. Often, the cases for the weakest precondition proofs via WP-LIFT-STEP include some assumption that $m_1 \preceq \sigma_1$ within the state interpretation. For our program logic, the points-to fragments in the premise of a Hoare law entail that m_1

"steps to" m_2. We then need to show safety: that there exists some σ_2 such that σ_1 steps to σ_2 (showing that there is a reduction); and preservation: that for any new σ_2 obtained from stepping from σ_1 that we may show $m_2 \preceq \sigma_2$ in order to reaffirm the state interpretation.

In our program logic, we may describe proofs using WP-LIFT-STEP as a *simulation*[7], which we illustrate in fig. 2.4. In particular, when invoking WP-LIFT-STEP in a context where we only perform a tree access without changing the tree structure, there are sufficient assumptions to construct m_2. We employ this style of diagram of fig. 2.4 in chapter 5 to explain important intricacies of weakest precondition proofs in our underlying model. As we expand our reasoning principles for Tree Borrows, the case analysis for preservation accumulates greater complexity.

2.2.4 Lambda Rust and Block-based Memory

Lambda Rust (λ_{Rust}) is a core calculus defined in Rocq intended to reflect the heap and ownership semantics of full Rust, and developed as part of the larger RustBelt [13] effort. λ_{Rust} possesses many important features intended to scale up to a larger Rust verification effort, such as data race protection in the heap. RustBelt also provides a *lifetime logic* in Iris, which features *borrow propositions* reflecting the borrow checker. RustBelt uses this lifetime logic to construct a proof of type safety for λ_{Rust} via a logical relation. Most of these aspects do not concern us here.

λ_{Rust}'s *block-based heap* is the main salient feature pertinent to this thesis. Block-based memory a la Compcert [19] is a technique to represent the physical memory layout in terms of blocks. λ_{Rust} defines the set of locations in the heap as $\mathit{Loc} \triangleq \mathit{BlockId} \times \mathbb{Z}$, where $\mathit{bid} : \mathit{BlockId}$ represents a *block identifier* and $z : \mathbb{Z}$ is some integer offset into the block. The type of the physical heap is essentially[8] $\mathit{State} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}$ for this notion of locations as offsets into blocks. Critically, λ_{Rust} *does not consider an aliasing model*, which is the primary difference between λ_{Rust} and our work. We provide further discussion comparing RustBelt and λ_{Rust} against our own work in section 6.4.

[7]We acknowledge that we may be abusing this terminology from state transition systems. Furthermore, we acknowledge that our notion of "step" is deterministic for laws strictly just performing accesses over the tree, not modifying its structure.

[8]We ignore the data race protection mechanisms in λ_{Rust}, which are irrelevant to our intents and purposes.

Chapter 3

Tree Library

In order to reason about a language with Tree Borrows, we require a robust library for trees. This chapter provides a birds-eye overview of important definitions and theorems describing our custom tree library.

3.1 Core Definitions

```
data Rose a = Node a [Rose a]
```

Figure 3.1: Typical Rose Tree data structure in Haskell, where nodes house their children in a list. See the Haskell Rose Tree library [4] for further implementation details.

We require a notion of trees in Rocq compatible with our separation logic in Iris. In separation logic, one often needs to break up, combine, and perform *frame-preserving updates* to resources. In our case, we need to be able change and break up the structure of the borrow trees. The notion of a *Rose Tree* (fig. 3.1) is well known in functional programming. Rose Trees are a concrete representation of general, finite trees where nodes store their children within a list, and a node may have any finite number of children. Tree Borrows certainly has need of general trees, and the MiniRust [5] implementation uses Rose Trees.

When performing basic tree operations such as lookups or insertions in trees, we *lookup at or insert into an address*. In the case of Rose Trees, the address space is lists of natural numbers $[k_0, k_1, \ldots, k_n] : \mathit{List}(\mathbb{N})$, where a number k_i at a particular index i of the list indicates to lookup the subtree at position k_i of the children list i nodes deep into the tree, after already having traversed the previous address elements. Every node/subtree has a unique address from the root, and we assign the root the empty address ϵ.

However, storing children within lists poses issues. Specifically, operations that modify the structure of the tree such as insertion and deletion *are not* *stable* for Rose Trees. For instance, inserting or deleting a tree from the middle of the children list changes the addresses of the subtrees in the rest of the list. Stability is an important property for our separation logic[1], so users need not contend with *address shifts* of other nodes in the tree if they need to perform retags[2], or wish to delete subtrees.

```
Unset Elimination Schemes.
Inductive tree
    `{Countable K} {A : Type} : Type :=
  (* Nodes. *)
  Tree {
      (* Data/label at a node. *)
      tree_data: A;
      (* Immediate subtrees. *)
      tree_children: gmap K tree }.
Set Elimination Schemes.
```

(a) Tree data type in Rocq.

$$t \in \mathit{Tree}(K, A) \triangleq \mathsf{Tree}(\left\{ \begin{array}{c} \textsc{data} : A, \\ \textsc{children} : K \xrightarrow{\text{fin}} \mathit{Tree}(K, A)) \end{array} \right\})$$

(b) Tree data type.

Figure 3.2: Lilac Tree data structure.

Enter Lilac Trees (fig. 3.2). In Lilac Trees[3], nodes store their children in a *finite map* with some domain K. In Rocq (fig. 3.2a), we use the gmap [17] data type from Rocq-std++ [25]. gmap satisfies important properties for our Lilac Trees.

- We may embed gmap into inductive data structures such as our Lilac Tree while satisfying Rocq's strict positivity criteria, and we may define a suitable induction principle for our trees.
- gmap satisfies *extensionality on Leibniz equality* with respect to lookups, which we may extend to our trees.

[1]This work does not yet support splitting trees into parent/child trees, which would definitely benefit from stable addresses. However, stability is still an important sanity property for retagging and ghost subtree deletion.

[2]The operational semantics of our core calculus generates new keys for tree addresses non-deterministically, preventing programs from forging provenance in the program logic.

[3]The name "Lilac Tree" is meant to be a nod to the name "Rose Tree". Roses nor lilacs grow in trees, but Rose Trees were allegedly named for the chaotic, recursive, and fractal growth of roses, and arguably lilac plants grow even more chaotically and fractally. Lilac Trees allow this more chaotic growth without compromising addresses. Furthermore, the flower naming scheme relates back to Iris. Most of this thesis will use "tree" instead of "Lilac Tree".

- gmap-based child storage enables address stability under insertion and deletion, since insertion and deletion in gmap leaves disjoint key-value pairs unchanged.

This thesis would not have been possible without gmap and its extensive libraries. The address space of Lilac Trees is $List(K)$ for any countable set K. We instantiate K with $\mathbb{N}$ for our Tree Borrows implementation, but our Lilac Tree library itself is flexible in principle.

```
(** Locate a subtree at address [ks] in [t].  *)
Global Instance tree_lookup `{Countable K} {A : Type} :
  Lookup (list K) (tree K A) (tree K A) :=
  fix go (ks : list K) (t : tree K A) : option (tree K A) :=
    let _ : Lookup _ _ _ := @go in
    match ks with
    | [] => Some t
    | k :: ks => (tree_children t) !! k >>= lookup ks
    end.
```

(a) Tree lookup in Rocq.

```
(** Insert subtree [st] at [ks] in [t].
    Silently fails if address is not valid.  *)
Global Instance tree_insert `{Countable K} {A : Type} :
  Insert (list K) (tree K A) (tree K A) :=
  fix go (ks : list K) (st t : tree K A) : tree K A :=
    let _ : Insert _ _ _ := @go in
    match ks with
    | [] => st
    | [k] => t <| tree_children ::= <[k := st]> |>
    | k :: ks => t <| tree_children ::= alter <[ks := st]> k |>
    end.
```

(b) Tree insertion in Rocq.

Figure 3.3: Primary lilac tree operations in Rocq.

$t \mathbin{!!} (ks_1 \mathbin{+\!\!+} ks_2) = t \mathbin{!!} ks_1 \gg\!= (. \mathbin{!!} ks_2)$	Decompose lookup.
$(\forall ks. (t_1 \mathbin{!!} ks).\textsc{data} = (t_2 \mathbin{!!} ks).\textsc{data}) \vdash t_1 = t_2$	Extensionality.
$\mathsf{removelast}(ks) \in t_1 \vdash t_1[ks \leftarrow t_2] \mathbin{!!} ks = \mathsf{Some}(t_2)$	Successful lookup.
$ks_1 \not\sqsubseteq ks_2 \vdash (t_2[ks_1 \leftarrow t_1] \mathbin{!!} ks_2).\textsc{data} = (t_2 \mathbin{!!} ks_2).\textsc{data}$	Stable insertion.

Figure 3.4: Basic tree lookup and insert lemmas, proved in Rocq.

Tree operations such as lookup (fig. 3.3a) and insertion (fig. 3.3b) are defined by recursion on the address. These operations satisfy useful properties as shown in fig. 3.4, especially *stable insertion and deletion*.

3.2 Relations on Trees

```
(** General relation between data of trees. *)
Definition tree_relation `{Countable K} {A B}
    (R : list K → A → B → Prop)
    (P : list K → A → Prop) (Q : list K → B → Prop)
    (ta : tree K A) (tb : tree K B) : Prop :=
  forall ks, option_relation (R ks) (P ks) (Q ks)
          (tree_data <$> ta !! ks) (tree_data <$> tb !! ks).
```

Figure 3.5: General tree relations.

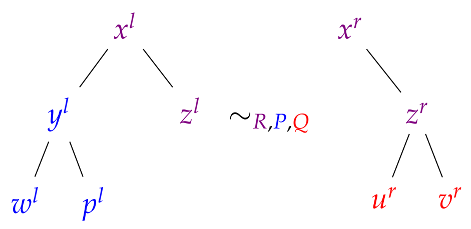


Figure 3.6: Visualization of general tree relations in fig. 3.5. We use $\sim$ to denote any general tree relation parameterized by some R, P, and Q. We denote shared nodes in violet, left-hand-side-only nodes in blue, and right-hand-side-only nodes in red. Relation R must hold for shared nodes, predicate P must hold for LHS-only nodes, and predicate Q must hold for RHS-only nodes.

We provide a library for relations between trees parameterized by pointwise relations between their elements. We give the formal Rocq definition in fig. 3.5, and a visualization in fig. 3.6. Via our general framework, we recover some important relations. If both P and Q are $\perp$, then we have a *pointwise relation R over trees with identical structure*. This is sufficient for *pointwise permission weakening* in the ghost tree relation $\preceq$, which we discuss in section 5.2. If just P is $\perp$, then we have *structural inclusion*, relating shared nodes by R, and constraining RHS-only nodes by Q. This is sufficient for *subtree deletion* in the ghost tree relation $\preceq$, which we discuss in section 5.3. If both P and Q are $\top$, then we have a *pointwise relation R over the trees with otherwise arbitrary structure*, or *agreement under R*. We require this for *protector agreement*, which we discuss in section 5.4 and section 5.4.2.

3.3 Monadic Transformations

Functor map and monadic transformations over trees play an important role in Tree Borrows. Often, these transformations need to be aware of the current address, such as when computing the locality of a Tree Borrows access. Since accesses may fail for individual Tree Borrows permissions, we require monadic operators to lift the monad over the entire tree data structure.

3.4 Tree Union

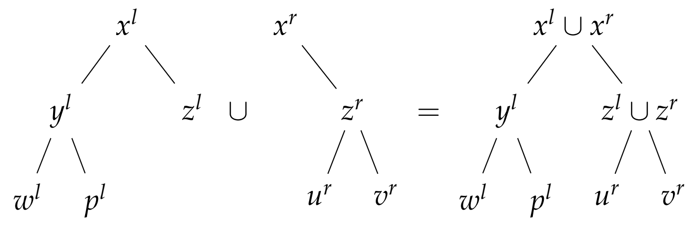


Figure 3.7: Visualization of tree union.

We provide a notion of union between trees. Even though we call it "union", this operation actually lifts a union operator on the data of the tree over the entire tree, and incorporates the structure of both trees. This tree union is *the operator* for lateral/blockwise separation of ghost trees. We present a visualization in fig. 3.7.

Chapter 4

The Program Logic Interface

In this chapter, we provide descriptions of the core calculus with tree borrows λ^{TB}, its syntax and structural operational semantics (SOS), and the program logic laws and user-facing lemmas.

4.1 A Core Calculus with Tree Borrows

Here we describe the syntax and operational semantics of our core calculus λ^{TB}, which intends to capture the core features of Rust related to Tree Borrows. λ^{TB} includes a similar block-based memory model to that of λ_{Rust} [13], but without the bells and whistles concerning data races. Overall, the syntax and semantics is essentially the same as that of HeapLang, but extended with Tree Borrows and block-based memory. The primary language features under consideration include:

- **Higher-order and first-class functions**: Rust's functional programming features interact with the aliasing model in nontrivial ways, such as references captured by closures, or opaque functions that may access the same memory locations as the calling context. Thus we would like λ^{TB} to capture these interactions to better reflect the realities of Rust programming.
- **Concurrency**: An important feature for many languages such as Rust and of course concurrent separation logics. λ^{TB} offers unstructured concurrency from a fork operator, from which we derive structured concurrency gadgets such as a parallel operator.
- **Block-based heaps**: As in λ_{Rust}, we choose to model the heap in terms of blocks of allocations. Consequently, during the physical execution, the semantics summons entire blocks of memory, not merely individual physical locations. One allocates an entire block in the heap, and then frees an entire block. Most heap accesses such as loads, stores,

and atomics only retrieve or change the values at individual locations, but nonetheless must operate at the granularity of entire blocks from physical memory. Tree Borrows is explicitly designed with blocks of contiguous memory in mind, such as how retags generate unaccessed permissions for parts of the block outside of the explicit retag range. Every node in the physical borrow trees includes permissions for the entire block. Furthermore, Tree Borrows assigns protectors at the level of nodes in borrow trees, not at every location in every node. Therefore, block-based memory is integral to our core calculus. Our language employs *sequential memory consistency*, and we leave any extensions for relaxed memory to future work. Our heaps are finite maps from block identifiers to blocks, which include the values and the borrow tree. We leave models that consider bytes and byte interpretation to future work.

- **Retagging**: Of course, aliasing is the primary behavior that an aliasing model such as Tree Borrows means to capture. Thus λ^{TB} needs to have an explicit notion of deriving new references (retagging). In a language with Tree Borrows, this entails adding a new node in the physical borrow tree with permissions for the entire block, even for locations outside of the explicit retag range. We require all retags to be syntactically explicit operations. In Tree Borrows [31], implicit retags for method and function application perform *protected retags*, where the new node is assigned a *protector*. Our core calculus only offers unprotected retags, and we leave protected[1] retags to future work. Furthermore, in Rust and Villani et al. [31][2] , the *retag range* may be determined at runtime, by arithmetic operations or opaque functions for instance. However, our core calculus mandates statically-known retag ranges, and dynamic sizes are left to future work.

It is with these core features in mind that we endeavor to capture aliasing under Tree Borrows in our core calculus and program logic. We leave to future work other Rust features that more tangentially interact with Tree Borrows such as parametric polymorphism, traits, more realistic memory models, more expressive algebraic data types, etc.

$$\begin{aligned}
\texttt{main}(()) = &\texttt{let}\, \mathit{mum} = \texttt{ref}(0)\;\texttt{in} \\
&\texttt{let}\, \mathit{boy} = \texttt{Retag}(\mathit{mum}, \texttt{mut})\;\texttt{in} \\
&\texttt{let}\, \mathit{girl} = \texttt{Retag}(\mathit{mum}, \texttt{mut})\;\texttt{in} \\
&\texttt{let}\, \mathit{grandson} = \texttt{Retag}(\mathit{girl}, \texttt{shared})\;\texttt{in} \\
&\mathit{girl} \leftarrow 99; \\
&\texttt{Free}(\mathit{girl})
\end{aligned}$$

Figure 4.1: Simple program from example 4 in λ^{TB}.

4.1.1 Syntax

We introduce our basic syntax by comparison in fig. 4.1, a λ^{TB} version of the Rust program from example 4. The syntax of λ^{TB} is reminiscent of HeapLang's, but with explicit reference derivations via `RetagN`.

We outline the syntax of λ^{TB} in fig. 4.2. Readers familiar with HeapLang will notice striking similarities. We support most features of HeapLang[3], such as basic lambda calculus terms (variables, functions, and function application) extended with language-primitive[4] recursive functions, unary and binary operators, basic product and sum data structures, a fork operator, and of course operations for reading from, writing to, allocating to, and freeing from the heap. Furthermore, we have an explicit `RetagN` operator to derive a new reference[5] from an existing location in memory. In $\texttt{RetagN}(e, n, rk)$, one provides a term e that evaluates to an existing reference, a statically-known retag size n, and a *reference kind* rk. As shown in fig. 4.2, a reference kind may indicate that the new reference is either mutable or shared/aliased/read-only.

[1]While we limit our core calculus and client-facing program logic to unprotected retags, our state machine and underlying ghost state for the program logic indeed feature protectors, as illustrated in section 5.2. As further described, there is still much work to be done to fully adopt protectors, but we have already laid a significant foundation.

[2]Villani et al. [31] includes both modifications to Miri itself with Tree Borrows, as well as a Rocq implementation and core calculus of its own for Simuliris. While the Miri Tree Borrows implementation supports dynamically-sized retags, their Rocq implementation also only supports statically-known retag sizes.

[3]HeapLang also has a notion of *prophecies* that we choose not to support for λ^{TB}.

[4]Here we mean to say that recursive functions are an explicit, primitive feature of the language, not that we only support *primitive recursion* in the computability theory sense.

[5]We also considered "pointer" instead of "reference", and perhaps "pointer" would better capture the intended meaning here, since this construct has a location in memory (a block identifier and offset) as well as *provenance*, additional state attached to the pointer, more than just the memory address. In our case, the provenance is the tag indicating the address in the borrow tree, which uniquely determines the reference. In order to appease proponents of "pointer" and "reference", we use the metavariable ρ for references in λ^{TB}, since Greek "ρ" is calligraphically similar to English "p".

$$
\begin{aligned}
\tau \in \mathit{Tags} &\triangleq \mathit{List}(\mathbb{N}) \\
\ell \in \mathit{Loc} &\triangleq \left\{ \begin{aligned} \textsc{block\_id} &: \mathit{BlockId}, \\ \textsc{offset} &: \mathbb{Z} \end{aligned} \right\} \\
\rho \in \mathit{Ref} &\triangleq \left\{ \begin{aligned} \textsc{block\_id} &: \mathit{BlockId}, \\ \textsc{offset} &: \mathbb{Z}, \\ \textsc{tag} &: \mathit{Tags} \end{aligned} \right\} \\
\mathit{rk} \in \mathit{RefKind} &::= \texttt{mut} \mid \texttt{shared} \\
v, w \in \mathit{Val} &::= z \mid \texttt{true} \mid \texttt{false} \mid () \mid \rho \mid \qquad\qquad (z \in \mathbb{Z}, n \in \mathbb{N}) \\
&\quad\ \texttt{rec}_{\texttt{v}}\ f(x) = e \mid (v, w)_{\texttt{v}} \mid \texttt{inl}_{\texttt{v}}(v) \mid \texttt{inr}_{\texttt{v}}(v) \\
e \in \mathit{Expr} &::= v \mid x \mid \texttt{rec}_{\texttt{e}}\ f(x) = e \mid e_1(e_2) \mid \\
&\quad\ \circledcirc_1 e \mid e_1 \circledcirc_2 e_2 \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
&\quad\ (e_1, e_2)_{\texttt{e}} \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid \texttt{inl}_{\texttt{e}}(e) \mid \texttt{inr}_{\texttt{e}}(e) \mid \\
&\quad\ \texttt{match}\ e\ \texttt{with inl} \Rightarrow e_1 \mid \texttt{inr} \Rightarrow e_2\ \texttt{end} \mid \\
&\quad\ \texttt{AllocN}(e_1, e_2) \mid \texttt{Free}(e) \mid \texttt{!}\,e \mid e_1 \leftarrow e_2 \mid \\
&\quad\ \texttt{CmpXchg}(e_1, e_2, e_3) \mid \texttt{Xchg}(e_1, e_2) \mid \texttt{FAA}(e_1, e_2) \mid \\
&\quad\ \texttt{RetagN}(e, n, \mathit{rk}) \mid \texttt{fork}\ \{e\} \\
\circledcirc_1 &::= - \mid \ldots \quad \text{(list incomplete)} \\
\circledcirc_2 &::= + \mid - \mid +_{\texttt{L}} \mid = \mid \ldots \quad \text{(list incomplete)}
\end{aligned}
$$

Figure 4.2: Syntax of λ^{TB}.

We intend this to mirror real Rust syntax, where references are shared by default, and the `mut` keyword informs the compiler that this is a mutable reference.

As depicted in fig. 4.2, a reference ρ is composed of a block identifier, an integer block offset (to allow pointer arithmetic), and a *tag*. A tag τ indicates the address in the borrow tree that uniquely determines the identity of the reference for some location in memory. As illustrated in fig. 4.2, a location in memory ℓ possesses a block identifier and an integer block offset just like a reference, but does not specify any particular reference with a tag. Unlike HeapLang and λ_{Rust}, locations are not included in the grammar of the language terms nor values themselves. Only references appear in the concrete syntax. This captures the intended behavior of Tree Borrows, where even *raw pointers* are not exempt from the restrictions of the aliasing model. A pointer in Rust under Tree Borrows possesses some *provenance* [29], which provides its tree address at runtime. Of course, explicit reference and pointer values with provenance do not occur in source Rust programs. As in many corecalculi for stateful languages such as HeapLang itself, we need the explicit references in the language syntax since we employ a *small-step operational semantics*. We discuss the generation of reference tags later on.

4.1.2 Operational Semantics

$$
\begin{aligned}
K \in \mathit{Ctx} &::= \bullet \mid \mathit{Ctx}_{>} \\
K_{>} \in \mathit{Ctx}_{>} &::= e(K) \mid K(v) \mid \\
&\quad \circledcirc_1 K \mid e \circledcirc_2 K \mid K \circledcirc_2 v \mid \texttt{if}\, K \,\texttt{then}\, e_1 \,\texttt{else}\, e_2 \mid \\
&\quad (e, K) \mid (K, v) \mid \texttt{fst}(K) \mid \texttt{snd}(K) \mid \\
&\quad \texttt{inl}(K) \mid \texttt{inr}(K) \mid \texttt{match}\, K \,\texttt{with inl} \Rightarrow e_1 \mid \texttt{inr} \Rightarrow e_2 \,\texttt{end} \mid \\
&\quad \texttt{AllocN}(e, K) \mid \texttt{AllocN}(K, v) \mid \texttt{Free}(K) \mid \texttt{!}\, K \mid e \leftarrow K \mid K \leftarrow v \mid \\
&\quad \texttt{CmpXchg}(e_1, e_2, K) \mid \texttt{CmpXchg}(e_1, K, v_3) \mid \texttt{CmpXchg}(K, v_2, v_3) \mid \\
&\quad \texttt{Xchg}(e, K) \mid \texttt{Xchg}(K, v) \mid \texttt{FAA}(e, K) \mid \texttt{FAA}(K, v) \mid \\
&\quad \texttt{RetagN}(K, n, rk)
\end{aligned}
$$

Figure 4.3: Evaluation contexts for λ^{TB}.

In this section, we outline the structural small-step operational semantics (SOS) of λ^{TB}. We employ a *contextual semantics*, which generally fits well with irisGS_gen, Iris's typeclass interface to instantiate the program logic and weakest precondition libraries. We present the evaluation contexts in fig. 4.3. Our evaluation contexts are essentially identical to HeapLang's, exhibiting a "right to left", strict evaluation order. We also add an evaluation context for `RetagN`. This enables pointer arithmetic used to determine the starting offset of the retag range to take place within the `RetagN` expression itself. Since the pure reduction semantics of λ^{TB} and HeapLang are virtually identical, we omit them and encourage the reader to view the Iris [27] documentation.

$$
\begin{aligned}
p \in \mathit{Permission} &\triangleq \mathsf{Reserved} \mid \mathsf{Unique} \mid \mathsf{Frozen} \mid \mathsf{Disabled} \\
ls \in \mathit{LocState} &\triangleq \mathbb{B} \times \mathit{Permission} \\
prot \in \mathit{Protection} &\triangleq \mathsf{Protected} \mid \mathsf{Unprotected} \\
blk \in \mathit{Block} &\triangleq \mathit{Vec}(\mathit{Val}, n) \times \mathit{Tree}(\mathbb{N}, \mathit{Protection} \times \mathit{Vec}(\mathit{LocState}, n)) \\
\sigma \in \mathit{State} &\triangleq \mathit{BlockId} \xrightarrow{\mathsf{fin}} \mathit{Block}^{?}
\end{aligned}
$$

Figure 4.4: Physical heap and Tree Borrows state machine permissions.

In fig. 4.4, we illustrate the construction of the physical heap $\sigma \in \mathit{State}$. Our heaps are finite maps from block identifiers *BlockId* to optional physical blocks $\mathit{Block}^?$, which include both a *vector* of values in the heap and a *physical tree* with a vector of *location states* at every node. An inhabited optional block $\mathit{Block}^?$ indicates a proper, allocated block. When an optional block is $\bot$, this indicates that the program freed the block from memory. The optional blocks $\mathit{Block}^?$ allow the operational semantics to always assign new block identifiers, and avoid reusing identifiers associated with previously deallocated blocks. Each node corresponds to a reference at that node's address, and the physical nodes contain the permissions at every location. Note that this organization for the heap is different from that of λ_{Rust} [13]. In λ_{Rust}, the heap is essentially[6] a finite map from locations (block identifiers and offsets) to values, $(\mathit{BlockId} \times \mathbb{Z}) \xrightarrow{\text{fin}} \mathit{Val}$. In contrast, our heap stratifies the block identifiers and offsets: one first looks up the block with the block identifier, and then the location within the block with the offset. We choose to stratify the heap structure in light of following considerations:

- Borrow trees organize nodes at the level of blocks, not individual locations. Recall that Tree Borrows assigns protector values for the entire block for a node, not individual locations. Also recall that retags insert new nodes with permissions for the entire block, not just for offsets within the retag range. Thus our heap should at the very least directly map block identifiers to borrow trees.
- By directly mapping block identifiers to blocks, we may pair the array of values with the borrow tree. As shown in fig. 4.4, there is some block size n used as the dependent index for a *vector of values* in the block, and we supply the same n for each *vector of location states* at every node in the borrow tree. Thus we unify the block sizes for both the value array and the nodes of the borrow tree, exploiting the power of dependent types to intrinsically maintain the representation invariants that *every node in the physical tree must have the same block size and offsets* and that *the nodes in the borrow tree must have identical block sizes and offsets to the value array.*

Now we turn our attention to the actual data within the physical borrow trees. We do not simply store a permission $p \in \mathit{Permission}$ at every offset in every node, but as already alluded to we store a *location state* $ls \in \mathit{LocState}$ at every offset in every node. As defined in fig. 4.4, a location state is a product of a boolean value representing an *accessed bit* and a Tree Borrows permission. The accessed bit indicates whether or not its offset in the node it inhabits in the borrow tree has been *locally accessed*. As further illustrated later on in section 5.2, the accessed bit primarily plays a role in the *protected permission*

[6] λ_{Rust} also maps every location to a *lock state* to reason about race conditions, but this is not relevant to this work at this time.

state machine, which when flipped on renders some accesses, especially foreign writes ($\uparrow$W), UB instead of Disabled, which prevents other code from interfering with this location. λ^{TB} does not currently support protected retags, but in order to future proof later work we include protectors[7] and accessed bits in our implementation of the Tree Borrows state machine and physical heap. Of course, we require the Tree Borrows permissions[8] themselves, and λ^{TB} supports the unprotected Tree Borrows state machine as shown in fig. 2.2.

Now we consider some of the main reduction rules for λ^{TB}, as detailed in fig. 4.5. We showcase the reduction laws for heap primitives such as loads $!\,e$, stores $e_1 \leftarrow e_2$, allocation, deallocation, and retagging. These primitives comprise the primary interface through which programs interact with Tree Borrows. We write $\mathsf{access}(acc, \tau, X, t_1) = t_2$ to denote that a borrow tree t_1 is accessed at tag/address τ and offsets X with an access type acc, and succeeds[9] in producing a new borrow tree t_2. Metavariable acc denotes whether we perform a read access R or a write access W. The astute reader may notice that the reduction rules generally adhere to a similar pattern:

1. We must always begin by looking up the block (a pair of values and a borrow, tree) in σ identified by some $\rho.\textsc{block\_id}$ (except of course for Allocate, which produces a fresh block).
2. We must ensure that the offset $\rho.\textsc{offset}$ is actually within the range of the block. In other words, the access must be within bounds.
3. We must ensure that the tag $\rho.\textsc{tag}$ is actually an address within the domain of the borrow tree. In order to access a reference ρ, this reference must veritably be a valid reference associated with some node in the borrow tree.
4. We must always perform a successful Tree Borrows access on the tree. This is a distinguishing trait of a program language semantics featuring Tree Borrows. We execute the state machine on every offset of every node in the borrow tree t_1, and upon success (no UB) we obtain a tree with updated permissions t_2. The state machine is only executed for location states within the accessed range of offsets, and the state

[7] As illustrated later on in chapter 5, protectors play an interesting role in the logical state for the program logic, and even their partial support incurs nontrivial design considerations. We hope that this foundation will aid in the complete addition of protectors to a Tree Borrows program logic.

[8] Observe that we do not yet support the *"Reserved Interior Mutable"* permission for *interior mutability*. See the conclusion discussing future work.

[9] A failing access corresponds to transitioning to UB ($\lightning$) in the Tree Borrows state machine. Thus in Rocq access is a *monadic* operation that produces an optional value: $\mathsf{Some}(t_2)$ upon success and None for failure/UB. In order to reduce notational clutter, we hide the optional and monadic operations for access in our presentation.

"Head" reduction (impure) $\boxed{e_1, \sigma_1 \rightarrow_{\mathsf{h}} e_2, \sigma_2}$

$$\textsc{Allocate}\quad \frac{\sigma(bid) = \bot \qquad 0 < z \qquad \rho = (bid, 0, \epsilon)}{(\texttt{AllocN}(z, v), \sigma) \rightarrow_{\mathsf{h}} (\rho, \sigma\left[bid \leftarrow (\vec{v}, \mathsf{Tree}(\overrightarrow{\mathsf{Unique}}, \emptyset))\right])}$$

$$\textsc{Free}\quad \frac{\begin{array}{c}\sigma(\rho.\textsc{block\_id}) = (\mathbf{v}, t) \qquad \rho.\textsc{offset} = 0 \\ \rho.\textsc{tag} \in \mathsf{dom}(t) \qquad \mathsf{access}(W, \rho.\textsc{tag}, \{0, \ldots, |\mathbf{v}| - 1\}, t) \neq \bot\end{array}}{(\texttt{Free}(\rho), \sigma) \rightarrow_{\mathsf{h}} ((), \sigma[\rho.\textsc{block\_id} \leftarrow \bot])}$$

$$\textsc{Load}\quad \frac{\begin{array}{c}\sigma(\rho.\textsc{block\_id}) = (\mathbf{v}, t_1) \qquad 0 \leq \rho.\textsc{offset} < |\mathbf{v}| \\ \rho.\textsc{tag} \in \mathsf{dom}(t_1) \qquad \mathsf{access}(R, \rho.\textsc{tag}, \rho.\textsc{offset}, t_1) = t_2\end{array}}{(!\,\rho, \sigma) \rightarrow_{\mathsf{h}} (\mathbf{v}(\rho.\textsc{offset}), \sigma[\rho.\textsc{block\_id} \leftarrow (\mathbf{v}, t_2)])}$$

$$\textsc{Store}\quad \frac{\begin{array}{c}\sigma(\rho.\textsc{block\_id}) = (\mathbf{v}, t_1) \qquad 0 \leq \rho.\textsc{offset} < |\mathbf{v}| \\ \rho.\textsc{tag} \in \mathsf{dom}(t_1) \qquad \mathsf{access}(W, \rho.\textsc{tag}, \rho.\textsc{offset}, t_1) = t_2\end{array}}{(\rho \leftarrow w, \sigma) \rightarrow_{\mathsf{h}} ((), \sigma[\rho.\textsc{block\_id} \leftarrow (\mathbf{v}[\rho.\textsc{offset} \leftarrow w], t_2)])}$$

$$\textsc{Retag}\quad \frac{\begin{array}{c}\sigma(\rho.\textsc{block\_id}) = (\mathbf{v}, t_1) \qquad 0 \leq \rho.\textsc{offset} \leq \rho.\textsc{offset} + n \leq |\mathbf{v}| \\ \rho.\textsc{tag} \in \mathsf{dom}(t_1) \qquad \rho.\textsc{tag} +\!\!+ [k] \notin \mathsf{dom}(t_1) \\ \mathsf{access}(R, \rho.\textsc{tag}, \{\rho.\textsc{offset}, \ldots, \rho.\textsc{offset} + n - 1\}, t_1) = t_2 \\ \rho' = (\rho.\textsc{block\_id}, \rho.\textsc{offset}, \rho.\textsc{tag} +\!\!+ [k])\end{array}}{(\texttt{RetagN}(\rho, n, rk), \sigma) \rightarrow_{\mathsf{h}} (\rho', \sigma\left[\rho.\textsc{block\_id} \leftarrow (\mathbf{v}, t_2\left[\rho.\textsc{tag} +\!\!+ [k] \leftarrow \mathsf{Tree}(\overrightarrow{[\![rk]\!]}, \emptyset)\right])\right])}$$

Figure 4.5: Impure reduction rules for λ^{TB}. These laws are similar to those of HeapLang, except that λ^{TB} employs block-based memory and of course features Tree Borrows, so these laws must perform accesses over borrow trees. We omit the reduction rules for fork, which is essentially identical to that from HeapLang, and for atomics which feature analogous interactions with the Tree Borrows state machine. Note that since λ^{TB} does not support protected retags, we elide the protector values and accessed bits to avoid clutter, and every protector value is implicitly Unprotected.

machine is run for every node in the tree. A node's address determines whether or not the accesses for its location states are local or foreign: the address is local for every node with an address *ks* that is a prefix $ks \sqsubseteq \rho.\textsc{tag}$ of the access address, otherwise the access is foreign. A failing access[10] even for a single location state in any node causes the entire heap operation to fail and renders it UB[11]. Thus these reduction laws must ensure that the accesses succeed *for every location state within range for every node*.

5. We must update the heap σ with the updated borrow tree t_2 (except for Free, where we deallocate the block entirely). Note that the borrow tree must be updated even for read accesses, which in general may change the permissions. This is a crucial difference between our λ^{TB} and HeapLang and even λ_{Rust}[12]. Any access to the physical state may modify the physical state in a programming language with Tree Borrows.

Now we will highlight the important, distinguishing components of our main reduction laws.

- Allocate: We non-deterministically obtain a fresh block identifier $bid \notin \mathsf{dom}(\sigma)$, and insert a new block indexed by *bid* with the given value v repeated z times, and a singleton borrow tree where every location state has a `true`[13] accessed bit and permission Unique. This remains the state of the root of the borrow tree, since *every access is local with respect to the root*. `AllocN` produces a new reference ρ assigned the fresh block identifier, offset 0, and an empty tag ϵ corresponding

[10]Locally writing to a Frozen permission, locally accessing a Disabled permission, ...

[11]Recall that a language definition with an aliasing model such as Tree Borrows incurs *more UB* than one without such a model. While more UB naively sounds undesirable, the more UB in a language the easier it is for compilers to perform optimizations. The trick is to balance UB so it is powerful enough for compilers, but does not create overwhelming obstacles and pitfalls for programmers, and in our case program logics. Compiler optimizations are out of scope for this work, so we refer the curious reader to Villani et al. [31] for further reference.

[12]The informed reader will likely point out that some read operations in λ_{Rust} over the heap do indeed update the physical state, specifically the notion of *lock state*. The point is that *some* read operations in λ_{Rust} do in fact leave the physical state unchanged, whereas *all* read operations in λ^{TB} update the physical state. Moreover, as we will illustrate in section 4.2, the changes in the physical state in λ^{TB} always manifest in changes in the logical state in the program logic, whereas many of the read Hoare laws for λ_{Rust} entirely hide any changes to the physical state. While it may be possible for λ^{TB} to eventually follow suit and conceal more of the borrow tree operations, it is unlikely that borrow tree accesses can be concealed entirely from clients of the program logic.

[13]A Unique location state must always possess a `true` accessed bit. A location state that is both unaccessed and Unique is an invalid state. Despite that this is an invariant of Tree Borrows, our development treats this more as an *emergent property* of Tree Borrows rather than a condition that must be explicitly upheld and inspected. This may play a more important rule in future work when protectors are fully adopted, since accessed bits primarily influence the protected state machine shown in fig. 5.3.

to the root of the tree. As we will further discuss in section 4.2, the non-deterministic generation of fresh block identifiers becomes *demonic* in our program logic.

- FREE: In order to free a block of memory, we must successfully perform a write access to every offset in the block. Note that the tag ρ.TAG need not refer to the root, and may denote an address arbitrarily deep in the borrow tree. This provides programs some flexibility: even though the Tree Borrows write access must not fail, one may use any reference that allows the write access to succeed. At the end, we deallocate[14] the block. Note that in the full Tree Borrows semantics, in order to perform a deallocation, the borrow tree *must not harbor any protected nodes*. We do not yet enforce this constraint since λ^{TB} does not yet support protected retags.

- LOAD: We perform a read access on the tree for just offset ρ.OFFSET, and update the heap with the new borrow tree.

- STORE: We perform a write access on the tree for just offset ρ.OFFSET, and update the heap with the new borrow tree, as well as with the new value.

- RETAG: In order to retag and generate a new reference, we must perform a successful read access on the existing borrow tree for the given range of offsets. Furthermore, we non-deterministically generate a new key k such that the address $\rho.\textsc{tag} \mathbin{+\!\!+} [k]$ is *not* already an existing address in the borrow tree, otherwise the reference would not be new. Recall from section 3.1 that our Lilac Trees enjoy stable insertion (and deletion). Let t_1' denote the subtree in t_1 at address ρ.TAG such that $t_1(\rho.\textsc{tag}) = t_1'$. We only require that $k \notin \mathsf{dom}(t_1'.\textsc{children})$. The stability of Lilac Trees allows the semantics to non-deterministically produce any such k. If λ^{TB} employed naive Rose Trees, then appending the new node at the end of the list of children would be the only natural course of action without further complicating the representation. As with ALLOCATE, this non-deterministic generation of a fresh k becomes demonic in the program logic. At the end of the retag operation, we return the new reference, and insert a new singleton tree at the new address. In our new singleton tree, we determine the permissions for every offset by converting the reference kind rk into the appropriate Tree Borrows permission via the denotation $[\![rk]\!]$. Recall the unprotected state machine diagram, illustrated in fig. 2.2. The entry point for a new mutable reference is the Reserved state, thus $[\![\texttt{mut}]\!] = \mathsf{Reserved}$. The entry point for a new read-only or shared reference is the Frozen state, thus $[\![\texttt{shared}]\!] = \mathsf{Frozen}$.

[14]In our informal presentation on paper, we do not explicitly show that the heap is assigned $\mathsf{Some}(\bot)$ at the block identifier. See the Rocq implementation for the full details.

The reduction laws for atomics are similar to those illustrated above, just with some extra steps. In Rocq, we have shown that λ^{TB} enjoys some basic metatheoretical properties, such as reduction preserving closedness (absence of free variables), and preservation of strictly positive block sizes. It remains future work to develop a type system or logical relation (such as the lifetime logic in Jung et al. [13]).

4.2 Program Logic

While λ^{TB} may share many similarities with HeapLang, our program logic for λ^{TB} must contend with the borrow trees. Accordingly, our separation logic for λ^{TB} requires a *borrow tree points-to* connective, which lamentably imposes additional complexity and onus upon clients of the logic. Nevertheless, our logic possesses powerful reasoning principles which assuage this burden.

4.2.1 A Grand Tour

We first illustrate the fundamentals of our program logic in fig. 4.6, a Hoare logic outline for fig. 4.1. Upon allocation we obtain points-tos for the block size (in this case simply 1), the value, and a singleton tree. Next, we perform a retag for a new mutable reference, adding a new reserved reference to the borrow tree for *boy*. We then perform another mutable retag, adding a new reserved reference for *girl*. Next, we perform an immutable retag for *grandson*, inserting a new Frozen node into the borrow tree. The ensuing write to *girl* disables the permissions for *boy* and *grandson*, and activates the permission for *girl*. Before deallocating memory, we *prune the subtrees for boy and grandson*, since we no longer locally access them. It would have also sufficed to delete them before the store to *girl*. Finally, we deallocate the memory, and relinquish ownership of our value and tree points-tos. Since block size points-tos are persistent, we may hold on to ours after the free. Note that in our proof in fig. 4.6, we rely upon concretely computing access to obtain the next borrow tree. But for specifications with universally quantified, arbitrary borrow trees, we require more reasoning principles.

4.2.2 Hoare Logic Laws for Tree Borrows

Here we detail the primitive, user facing Hoare logic laws. We require some points-to fragment associating locations in memory to ghost trees. Recall from fig. 4.4 that borrow trees in the physical state contain a vector of location states at every node. These vectors share a common length representing the block size. In order to assuage *lateral separation*, we require a more flexible representation. We show in fig. 4.7 the Lilac Tree instantiations we employ for the logical trees. We support both *block-spanning* ghost trees or *ghost*

$\{\top\}$
$\texttt{let}\ mum = \texttt{ref}(0)\ \texttt{in}$
$\{bid \mapsto_s 1 * (bid,0) \mapsto 0 * (bid,0) \mapsto_t \mathsf{Unique}\}$
$\texttt{let}\ boy = \texttt{Retag}(mum, \texttt{mut})\ \texttt{in}$

$$\left\{ bid \mapsto_s 1 * (bid,0) \mapsto 0 * (bid,0) \mapsto_t \begin{tikzcd} & \mathsf{Unique} \arrow[dl, no head] \\ \mathsf{Reserved} & \end{tikzcd} \right\}$$

$\texttt{let}\ girl = \texttt{Retag}(mum, \texttt{mut})\ \texttt{in}$

$$\left\{ bid \mapsto_s 1 * (bid,0) \mapsto 0 * (bid,0) \mapsto_t \begin{tikzcd} & \mathsf{Unique} \arrow[dl, no head] \arrow[dr, no head] & \\ \mathsf{Reserved} & & \mathsf{Reserved} \end{tikzcd} \right\}$$

$\texttt{let}\ grandson = \texttt{Retag}(girl, \texttt{shared})\ \texttt{in}$

$$\left\{ bid \mapsto_s 1 * (bid,0) \mapsto 0 * (bid,0) \mapsto_t \begin{tikzcd} & \mathsf{Unique} \arrow[dl, no head] \arrow[dr, no head] & \\ \mathsf{Reserved} & & \mathsf{Reserved} \arrow[d, no head] \\ & & \mathsf{Frozen} \end{tikzcd} \right\}$$

$girl \leftarrow 99;$

$$\left\{ bid \mapsto_s 1 * (bid,0) \mapsto 99 * (bid,0) \mapsto_t \begin{tikzcd} & \mathsf{Unique} \arrow[dl, no head] \arrow[dr, no head] & \\ \mathsf{Disabled} & & \mathsf{Unique} \arrow[d, no head] \\ & & \mathsf{Disabled} \end{tikzcd} \right\}$$

$$\left\{ bid \mapsto_s 1 * (bid,0) \mapsto 99 * (bid,0) \mapsto_t \begin{tikzcd} \mathsf{Unique} \arrow[dr, no head] & \\ & \mathsf{Unique} \end{tikzcd} \right\}$$

$\texttt{Free}(girl)$
$\{bid \mapsto_s 1\}$

Figure 4.6: Hoare logic proof for fig. 4.1.

$\mathbf{t} \in Tree(\mathbb{N}, Protection \times (\mathbb{N} \xrightharpoonup{\text{fin}} LocState))$ Block-spanning ghost tree/grove.
$t \in Tree(\mathbb{N}, Protection \times LocState)$ Single-location ghost tree.

Figure 4.7: Ghost tree definitions. We denote a block-spanning ghost tree or *grove* that may contain multiple offsets as **t**, and we denote a ghost tree for a single location as t.

groves[15] that may possess multiple, continuous offsets in the block, as well as *per-location* ghost trees that represent the permissions just for a single location in memory. The reader concerned with low-level memory semantics and byte representation may be inclined to point out that such single-location ghost trees would not be user-friendly in a program logic where every memory location maps to a single byte, and where memory accesses read from and write to spans of bytes. This is a fair criticism, but for our core calculus that simply stores entire arbitrary values at each location, such a per-location tree is useful when stepping through loads and stores, which in λ^{TB} only access a single location. As we will explain in section 5.4.1, the block-spanning ghost trees do not require the same set of offsets at every node, but require the set of offsets to be *well-formed* fig. 5.10. Thus in future work that tackles more realistic memory layouts with sized types and accesses to multiple bytes at a time, it may be prudent to introduce a constrained version of block-spanning trees that feature the same set of offsets at every node.

In fig. 4.8, we detail some of the primitive[16] laws for our logic, and in fig. 4.9, we exhibit some of main laws our points-tos enjoy. We express laws for primitives that work with the entire block or multiple-offset spans of the block (WP-AllocN, WP-Free, WP-RetagN) in terms of block-spanning ghost groves, whereas we express laws for primitives that access just a single location (WP-Load, WP-Store) in terms of per-location ghost trees. Laws expressed in terms of ghost groves need more guarantees about the structure of the trees. As shown for WP-Free and WP-RetagN, the node inhabiting the access address/tag must contain all offsets directly read from or written to. This appears more complicated when compared to the assumptions needed for Free and Retag, which do not need to explicitly lookup the node at the access address, and instead just check that the access address is within the tree's domain. We need the explicit lookup and offset membership assumptions for WP-Free and WP-RetagN but not for Free and Retag because the ghost groves in the primitive laws structure their block fragments in finite maps, whereas in the reduction rules the dependent index for the block size constrains every node in each physical tree to have the same, full set of block offsets at every node. In short, we require these more detailed assumptions to curb the degrees of freedom inherit to the representation of ghost groves as shown in fig. 4.7.

However, by virtue of their construction, laws expressed in terms of per-location trees merely need to confirm that the tag is a valid address in the tree.

[15]In plain English, a "grove" is a small group of trees. We deem this appellation appropriate since these block-spanning logical trees entail the ownership of the physical tree for multiple locations. Furthermore, as will be later explored in section 5.4, these block-spanning trees are merely an interface for a continuous group of per-location ghost trees, which renders the epithet "grove" all the more apropos.

[16]WP-Par is actually a derived law from that of `fork` $\{e\}$ (not shown).

Hoare triple $\boxed{\{P\}\ e\ \{w.Q\}}$

WP-AllocN

$$\{0 < z\}$$
$$\texttt{AllocN}(z, v)$$
$$\left\{w.\exists bid. w = (bid, 0, \epsilon) * bid \mapsto_s z * (bid, 0) \mapsto\!\!* \vec{v} * (bid, 0) \mapsto\!\!*_t \mathsf{Tree}(\vec{\mathsf{Unique}}, \emptyset)\right\}$$

WP-Free

$$\frac{\begin{array}{c}\mathbf{t}(\tau) = \mathbf{t}_\tau \\ \mathsf{dom}(\mathbf{t}_\tau.\textsc{data}) = \{0, \ldots, |\mathbf{v}| - 1\} \qquad \mathsf{access}(W, \tau, \{0, \ldots, |\mathbf{v}| - 1\}, \mathbf{t}) \neq \bot\end{array}}{\{bid \mapsto_s |\mathbf{v}| * (bid, 0) \mapsto\!\!* \mathbf{v} * (bid, 0) \mapsto\!\!*_t \mathbf{t}\}\ \texttt{Free}(bid, 0, \tau)\ \{().\top\}}$$

WP-Load

$$\frac{\tau \in \mathsf{dom}(t_1) \qquad \mathsf{access}(R, \tau, z, t_1) = t_2}{\left\{(bid, z) \overset{dq}{\mapsto} v * (bid, z) \mapsto_t t_1\right\}\ !(bid, z, \tau)\ \left\{v.(bid, z) \overset{dq}{\mapsto} v * (bid, z) \mapsto_t t_2\right\}}$$

WP-Store

$$\frac{\tau \in \mathsf{dom}(t_1) \qquad \mathsf{access}(W, \tau, z, t_1) = t_2}{\{(bid, z) \mapsto v * (bid, z) \mapsto_t t_1\}\ (bid, z, \tau) \leftarrow w\ \{().(bid, z) \mapsto w * (bid, z) \mapsto_t t_2\}}$$

WP-RetagN

$$\frac{\begin{array}{c}\mathbf{t}_1(\tau) = \mathbf{t}'_1 \qquad z \in \mathsf{dom}(\mathbf{t}'_1.\textsc{data}) \qquad \{z, \ldots, z + n - 1\} \subseteq \mathsf{dom}(\mathbf{t}'_1.\textsc{data}) \\ \mathsf{access}(R, \tau, \{z, \ldots, z + n - 1\}, \mathbf{t}_1) = \mathbf{t}_2\end{array}}{\begin{array}{l}\{(bid, z) \mapsto\!\!*_t \mathbf{t}_1\} \\ \quad \texttt{RetagN}((bid, z, \tau), n, rk) \\ \left\{w.\exists k. w = (bid, z, \tau \mathbin{+\!\!+} [k]) * \tau \mathbin{+\!\!+} [k] \notin \mathsf{dom}(\mathbf{t}_2) * (bid, z) \mapsto\!\!*_t \mathbf{t}_2\left[\tau \mathbin{+\!\!+} [k] \leftarrow \mathsf{Tree}(\llbracket \vec{rk} \rrbracket, \emptyset)\right]\right\}\end{array}}$$

WP-Par

$$\frac{\{P_1\}\ e_1\ \{v_1.\ \Phi_1(v_1)\} \qquad \{P_2\}\ e_2\ \{v_2.\ \Phi_2(v_2)\}}{\{P_1 * P_2\}\ e_1 \mid\mid e_2\ \{w.\ \exists v_1 v_2.\ w = (v_1, v_2) * \Phi_1(v_1) * \Phi_2(v_2)\}}$$

Figure 4.8: Primitive, user-facing Hoare logic laws for λ^{TB}. Since the operational semantics of λ^{TB} has yet to properly support protected retags, we elide the details of protector values and accessed bits here. For the full details, see the Rocq artifact.

Block-size-Persistent

$$bid \mapsto_s n \vdash \Box\, bid \mapsto_s n$$

Block-size-Positive

$$bid \mapsto_s n \vdash 0 < n$$

Block-size-Agree

$$bid \mapsto_s n_1 * bid \mapsto_s n_2 \vdash n_1 = n_2$$

Value-offset-Non-negative

$$\ell \overset{dq}{\longmapsto} v \vdash 0 \leq \ell.\textsc{offset}$$

Values-Agree

$$\ell \overset{q_1}{\longmapsto} v * \ell \overset{q_2}{\longmapsto} w \vdash \ell \overset{q_1+q_2}{\longmapsto} v * 0 < q_1 + q_2 \leq 1 \land v = w$$

Value-offset-within-Block

$$\frac{\ell.\textsc{block\_id} = bid}{bid \mapsto_s n * \ell \overset{dq}{\longmapsto} v \vdash \ell.\textsc{offset} < n}$$

Append-Values

$$\ell \overset{dq}{\longmapsto\!\!*} \mathbf{v} +\!\!+ \mathbf{w} \dashv\vdash \ell \overset{dq}{\longmapsto\!\!*} \mathbf{v} * \ell +_{\mathrm{L}} |\mathbf{v}| \overset{dq}{\longmapsto\!\!*} \mathbf{w}$$

Value-Accessor

$$\frac{\mathbf{v}(z) = v}{\ell \overset{dq}{\longmapsto\!\!*} \mathbf{v} \vdash \ell +_{\mathrm{L}} z \overset{dq}{\longmapsto} v * \forall w, \ell +_{\mathrm{L}} z \overset{dq}{\longmapsto} w \mathrel{-\!\!*} \ell \overset{dq}{\longmapsto\!\!*} \mathbf{v}[z \leftarrow w]}$$

Tree-offset-Non-negative

$$\ell \mapsto_t t \vdash 0 \leq \ell.\textsc{offset}$$

Tree-offset-within-Block

$$\frac{\ell.\textsc{block\_id} = bid}{bid \mapsto_s n * \ell \mapsto_t t \vdash \ell.\textsc{offset} < n}$$

Weaken-Tree

$$\frac{t_1 \preceq t_2}{\ell \mapsto_t t_2 \vdash \ell \mapsto_t t_1}$$

Tree-to-Grove

$$\ell \mapsto_t t \dashv\vdash \ell \mapsto\!\!*_t \mathcal{T}[\![t]\!]_{\ell.\textsc{offset}}$$

Grove-Well-formed

$$\ell \mapsto\!\!*_t \mathbf{t} \vdash \mathsf{dom}(\mathbf{t}.\textsc{data}) = \{\ell.\textsc{offset}, \ldots, \ell.\textsc{offset} + |\mathsf{dom}(\mathbf{t}.\textsc{data})| - 1\} \land \mathsf{WF}(\mathbf{t})$$

Combine-Grove

$$\frac{\ell_2 = \ell_1 +_{\mathrm{L}} |\mathsf{dom}(\mathbf{t}_1.\textsc{data})|}{\ell_1 \mapsto\!\!*_t \mathbf{t}_1 * \ell_2 \mapsto\!\!*_t \mathbf{t}_2 \vdash \ell_1 \mapsto\!\!*_t \mathbf{t}_1 \cup \mathbf{t}_2}$$

Split-Grove

$$\frac{\begin{array}{c}\ell_2 = \ell_1 +_{\mathrm{L}} |\mathsf{dom}(\mathbf{t}_1.\textsc{data})| \\ \mathsf{WF}(\mathbf{t}_1) \qquad \mathsf{WF}(\mathbf{t}_2) \qquad \mathsf{dom}(\mathbf{t}_1.\textsc{data}) \perp \mathsf{dom}(\mathbf{t}_2.\textsc{data}) \\ \mathsf{dom}(\mathbf{t}_1.\textsc{data}) = \{\ell_1.\textsc{offset}, \ldots, \ell_1.\textsc{offset} + |\mathsf{dom}(\mathbf{t}_1.\textsc{data})| - 1\}\end{array}}{\ell_1 \mapsto\!\!*_t \mathbf{t}_1 \cup \mathbf{t}_2 \vdash \ell_1 \mapsto\!\!*_t \mathbf{t}_1 * \ell_2 \mapsto\!\!*_t \mathbf{t}_2}$$

Tree-Accessor

$$\ell \mapsto\!\!*_t \mathbf{t} \vdash (\ell.\textsc{block\_id}, i) \mapsto_t \mathbf{t}[i] * \forall t, (\ell.\textsc{block\_id}, i) \mapsto_t t \mathrel{-\!\!*} \ell \mapsto\!\!*_t t \cup \mathbf{t} \setminus \mathbf{t}[i]$$

Figure 4.9: User-facing points-to laws for λ^{TB}.

Thus WP-Load and WP-Store simply require $\tau \in \mathsf{dom}(t)$. These laws thus yield less proof overhead than WP-Free and WP-RetagN. Allocating and retagging from a single location is also a common pattern, thus in Rocq we provide derived laws for these cases. In Rocq, we also have derived laws for the ghost grove case for loads and stores to indulge users that would prefer a single exposed logical representation of the borrow trees. Some potential users may reject or be overwhelmed by the seemingly competing logical representations of borrow trees, as shown in fig. 4.7. While featuring only one logical representation may appear to be more accessible, this would actually make the logic more burdensome to use. If the interface only had per-location trees, then retagging would involve a list of these trees, and one would need to map an insert operation over the list of trees. Furthermore, the assumption for a successful Tree Borrows access would need to be carried over the list of trees, rather than computing a single ghost grove for a successful access. If the interface only featured block-spanning ghost trees, then laws for per-location operations such as WP-Load and WP-Store would also require the explicit node lookup and block offset membership assumptions that WP-Free and WP-RetagN do, which is especially burdensome in the case of a single offset allocation. In Rocq, we perform proofs in both styles, and the two logical representations offer *flexibility and convenience*, and shorter, less tedious proof scripts. Our model has some limitations concerning conversions between points-tos for per-location trees and single-location groves, which we discuss in section 6.2.

Note that WP-AllocN produces and WP-Free requires a points-to assertion for the block size. As shown in fig. 4.9, this is a persistent resource that may be combined with those for values and trees to enforce that the offsets owned by value or tree points-to are within the block. As is true of λ_{Rust}, the block size points-to is only needed for WP-Free to enforce ownership of the values and borrow tree for the entire block.

In WP-AllocN and WP-RetagN, note that new block identifiers and tree addresses respectively are *demonically non-deterministic*: the new identifiers and addresses are not deterministically computed nor generated, and program logic users *do not supply* their new values, but instead *receive any possibe value*. This is a consequence of the operational semantics rules Allocate and Retag, as well as the definition of Iris's notion of weakest preconditions. This precludes programs under verification from anticipating specific values of new block identifiers and tree addresses, and their demonically non-deterministic generation engenders the *unforgeability of block identifiers and tags* under the program logic. Forging pointer provenance would both violate Tree Borrows itself and potentially disrupt the correctness of compiler optimizations.

Now let us turn our attention to the points-to laws as shown in fig. 4.9.

Laws such as Value-offset-Non-negative and Tree-offset-Non-negative demonstrate that the offset of the points-to location is at least 0, and laws such as Value-offset-within-Block and Tree-offset-within-Block use the block size points-to to enforce that the offset is less than the block size, constraining the that the offset is within bounds. Block-spanning trees or groves enjoy a similar in-bounds restriction. Also note that the points-tos for a single value and a list of values enjoy the typical separation properties such as Values-Agree, Append-Values, and Value-Accessor that the analogous constructs enjoy for HeapLang and λ_{Rust}. These laws allow users to break apart ownership across the block for values, and with Value-Accessor clients may extract ownership of a single value within the block, perform updates, and then snap it back in.

Like values, points-tos for block-spanning trees enjoy similar properties. First, note that grove ownership via Grove-Well-formed entails that the root domain of the block-spanning tree is some continuous set of offsets starting from the points-to's location's offset. Laws such as Combine-Grove and Split-Grove represent *adjacent lateral merging and separation* of block-spanning trees/groves. These laws are the tree analogue to Append-Values. However, laterally splitting trees (Split-Grove) requires more assumptions, especially the well-formedness of each component via $\mathsf{WF}(\mathbf{t})$, which we detail later in section 5.4.1. Note that unlike values, offsets in groves are *global*[17] , not relative with respect to the points-to's location. As with Value-Accessor, we may borrow out a single-location tree points-to and plug it back into the full original grove points-to via Tree-Accessor. We may also convert between a single-location tree t points-to and its injection into a single-location grove $\mathcal{T}[\![t]\!]_{\ell.\text{OFFSET}}$ via Tree-to-Grove.

The law Weaken-Tree allows one to exchange a points-to with t_2 for one with a *logically weaker* ghost tree t_1. The relation $t_1 \preceq t_2$ entails both pointwise permission weakening and subtree deletion, which we discuss in more detail in section 5.2 and section 5.3 respectively.

4.2.3 Permission Weakening Example

Now, we consider an example program where a function accepts a reference as a parameter, and where we must reconcile the permissions in the final borrow resulting from different execution branches. Regard fig. 4.10. Function `foo` accepts an opaque, boolean function *opaque*, which it uses to determine whether or not to write to a reference assigned to x. While we may allow one to possess a disjunction of the final tree permissions, we may also *logically* weaken our borrow tree from the read branch.

[17]Global offsets in our block-spanning groves is indeed an limitation of our design. The issue lies in constructing witnesses for lateral tree separation. See section 6.2 for further discussion.

$$\begin{aligned}\texttt{foo}(\mathit{opaque}, x) = {} & \texttt{let}\, r = \texttt{Retag}(x, \texttt{mut})\ \texttt{in} \\ & \texttt{if}\, \mathit{opaque}(())\ \texttt{then}\, r \leftarrow 42 \\ & \texttt{else}\ !r\end{aligned}$$

(a) Program `foo` accepts some opaque boolean function and a reference, and writes to or reads from the reference depending upon the value of the opaque function.

$$\textsc{Foo-Spec}\quad \frac{\begin{array}{c}\{\top\}\, \mathit{opaque}(())\, \{b.\, b = \texttt{true} \lor b = \texttt{false}\} \\ \ell = (\mathit{bid}, z) \qquad \tau \in \mathsf{dom}(t_1) \qquad \mathsf{access}(W, \tau, z, t_1) = t_4\end{array}}{\begin{array}{l}\{\ell \mapsto v * \ell \mapsto_t t_1\} \\ \quad \texttt{foo}(\mathit{opaque}, (\mathit{bid}, z, \tau)) \\ \left\{ \exists k.\, \tau +\!\!+ [k] \notin \mathsf{dom}(t_4) * (\ell \mapsto 42 \lor \ell \mapsto v) * \ell \mapsto_t \begin{array}{c} t_4 \\ \vdots\ \tau +\!\!+ [k] \\ \mathsf{Unique} \end{array} \right\}\end{array}}$$

(b) Specification for `foo`.

Figure 4.10: Program `foo` in λ^{TB}

We outline our proof for `foo` in fig. 4.11. Our proof requires the assumptions that *opaque* is some boolean function, as well as ownership of the value and tree accessed and written to. However, if we try to apply WP-RetagN for the first retag, we immediately realize that merely owning some arbitrary tree is not enough. Unlike fig. 4.11, our ghost tree is *arbitrary*, therefore we need to know that a read access to τ on this arbitrary tree succeeds. In fact, such an assumption is reasonable to add to our specification in Foo-Spec. Tree Borrows accesses fail in many cases, so explicitly assuming that the access succeeds and produces some new tree accurately reflects the operational semantics, where the tree is updated for every heap reduction rule.

Unfortunately, simply knowing that a read access succeeds is insufficient, since the "then" case performs a write. Furthermore, `foo` performs accesses one node deeper into the tree, not at the given address τ. Since tree addresses are demonically non-deterministically generated, the precondition of the specification for `foo` cannot specifically say that the write access succeeds at that address since we do not yet know what the new address will be. We would like some means of writing a specification for `foo` that is both *reasonably succinct* (avoids assuming multiple access results) and *sufficiently powerful* to carry us through the proof.

Fortunately, we may resolve this dilemma. Accesses in Tree Borrows enjoy

Context: $\{\top\}\ \mathit{opaque}(())\ \{b.\ b = \texttt{true} \vee b = \texttt{false}\}$

$$\{\mathsf{access}(W, \tau, \ell.\textsc{offset}, t_1) = t_4 * \ell \mapsto v * \ell \mapsto_t t_1\}$$

$$\left\{\begin{array}{l}\mathsf{access}(R, \tau, \ell.\textsc{offset}, t_1) = t_2 * \mathsf{access}(W, \tau, \ell.\textsc{offset}, t_2) = t_4 * \\ \ell \mapsto v * \ell \mapsto_t t_1\end{array}\right\}$$

$\texttt{let}\ r = \texttt{Retag}(x, \texttt{mut})\ \texttt{in}$

$$\left\{\begin{array}{l}\mathsf{access}(W, \tau, \ell.\textsc{offset}, t_2) = t_4 * \\ \ell \mapsto v * \ell \mapsto_t \overset{t_2}{\underset{\text{Reserved}}{\vdots}}\ \tau +\!\!+ [k]\end{array}\right\}$$

$\texttt{if}\ \mathit{opaque}(())\ \texttt{then}$

$$\left\{\mathsf{access}(W, \tau, \ell.\textsc{offset}, t_2) = t_4 * \ell \mapsto v * \ell \mapsto_t \overset{t_2}{\underset{\text{Reserved}}{\vdots}}\ \tau +\!\!+ [k]\right\}$$

$r \leftarrow 42$

$$\left\{\ell \mapsto 42 * \ell \mapsto_t \overset{t_4}{\underset{\text{Unique}}{\vdots}}\ \tau +\!\!+ [k]\right\}$$

$\texttt{else}$

$$\left\{\begin{array}{l}\mathsf{access}(R, \tau, \ell.\textsc{offset}, t_2) = t_3 * \mathsf{access}(W, \tau, \ell.\textsc{offset}, t_3) = t_4 * \\ \ell \mapsto v * \ell \mapsto_t \overset{t_2}{\underset{\text{Reserved}}{\vdots}}\ \tau +\!\!+ [k]\end{array}\right\}$$

$!\,r$

$$\left\{\mathsf{access}(W, \tau, \ell.\textsc{offset}, t_3) = t_4 * \ell \mapsto v * \ell \mapsto_t \overset{t_3}{\underset{\text{Reserved}}{\vdots}}\ \tau +\!\!+ [k]\right\}$$

$$\left\{\ell \mapsto v * \ell \mapsto_t \overset{t_4}{\underset{\text{Unique}}{\vdots}}\ \tau +\!\!+ [k]\right\}$$

$$\left\{(\ell \mapsto 42 \vee \ell \mapsto v) * \ell \mapsto_t \overset{t_4}{\underset{\text{Unique}}{\vdots}}\ \tau +\!\!+ [k]\right\}$$

Figure 4.11: Hoare logic proof for the conditional program from fig. 4.10.

$$
\begin{array}{rcll}
\mathsf{access}(acc,\tau,\emptyset,\mathbf{t}) &=& \mathbf{t} & \text{Access no offsets.}\\
\mathsf{access}(acc,\tau,Y,\mathsf{access}(acc,\tau,X,\mathbf{t})) &=& \mathsf{access}(acc,\tau,X\cup Y,\mathbf{t}) & \text{Decompose offsets.}\\
\mathsf{access}(acc,\tau,X,\mathsf{access}(acc,\tau,X,\mathbf{t})) &=& \mathsf{access}(acc,\tau,X,\mathbf{t}) & \text{Idempotency.}\\
\mathsf{access}(acc,\tau,X,\mathsf{access}(R,\tau,X,\mathbf{t})) &=& \mathsf{access}(acc,\tau,X,\mathbf{t}) & \text{Insert prior read.}\\
\mathsf{access}(R,\tau,X,\mathsf{access}(acc,\tau,X,\mathbf{t})) &=& \mathsf{access}(acc,\tau,X,\mathbf{t}) & \text{Append read.}
\end{array}
$$

$$
\textsc{Disjoint-Offsets}\quad \frac{X \perp \mathsf{dom}(\mathbf{t}) \qquad \mathsf{WF}(\mathbf{t})}{\mathsf{access}(acc,\tau,X,\mathbf{t}) = \mathbf{t}}
\qquad\qquad
\textsc{Weaken-via-Access}\quad \frac{\mathsf{access}^*(acc,\tau,X,\mathbf{t}_2) = \mathbf{t}_1}{\mathbf{t}_1 \preceq \mathbf{t}_2}
$$

$$
\textsc{Lateral-Decomposition}\quad \frac{\begin{array}{c}\mathsf{dom}(\mathbf{t}_1.\textsc{data}) \perp \mathsf{dom}(\mathbf{t}_2.\textsc{data})\\ \mathsf{WF}(\mathbf{t}_1) \qquad \mathsf{WF}(\mathbf{t}_2) \qquad \text{protectors agree for } \mathbf{t}_1 \text{ and } \mathbf{t}_2\end{array}}{\mathsf{access}(acc,\tau,X,\mathbf{t}_1\cup\mathbf{t}_2) = \mathsf{access}(acc,\tau,X,\mathbf{t}_1)\cup\mathsf{access}(acc,\tau,X,\mathbf{t}_2)}
$$

$$
\textsc{Commute-Insertion}\quad \frac{\begin{array}{c}\tau \mathbin{+\!\!+} [k] \notin \mathsf{dom}(\mathbf{t}_1)\\ \mathsf{access}(acc,\tau,X,\mathbf{t}_1) = \mathbf{t}_1' \qquad \mathsf{access}_{\tau \mathbin{+\!\!+} [k]}(acc,\tau \mathbin{+\!\!+} [k],X,\mathbf{t}_2) = \mathbf{t}_2'\end{array}}{\mathsf{access}(acc,\tau \mathbin{+\!\!+} [k],X,\mathbf{t}_1[\tau \mathbin{+\!\!+} [k] \leftarrow \mathbf{t}_2]) = \mathbf{t}_1'[\tau \mathbin{+\!\!+} [k] \leftarrow \mathbf{t}_2']}
$$

Figure 4.12: Reasoning principles for Tree Borrows accesses.

many beneficial properties, as shown in fig. 4.12. We may insert[18] read accesses before and after a write access, as well as commute an access with an insertion via Commute-Insertion[19].

The laws in fig. 4.12 equip us with the reasoning principles needed to specify and verify `foo`. Thus, as shown in fig. 4.11, we obtain evidence for a successful read access and dispatch the retag.

We must consider each case of the conditional program from fig. 4.10. First, we tackle the "then" case. While we possess knowledge that a write access is indeed successful, upon closer inspection of WP-Store one finds that we require knowledge of a successful write *for an access to tag* $\tau \mathbin{+\!\!+} [k]$, but we appear to only know that the write succeeds for tag τ. We apply Commute-Insertion to deduce that the write will succeed for the new tree resulting from the previous insertion. Thus we dispatch the write and conclude this case.

[18]Idempotency, read insertion and read appendment also hold for single-location ghost trees.

[19]The access assumption for the inserted tree $\mathbf{t}_2$ has a subscript $\tau \mathbin{+\!\!+} [k]$ to indicate that we perform the access at that starting depth. By default, tree accesses start with an empty depth ϵ signifying that the access is anchored at the root.

Now we consider the "else" case. Once again, we must insert a read access before the write, and apply COMMUTE-INSERTION to satisfy the access assumptions for WP-LOAD. After dispatching the load, we are left with a discrepancy: our tree after WP-LOAD does not match the tree expected by the postcondition. Fortunately, we may *logically weaken* the permissions of our borrow tree via WEAKEN-TREE. We may weaken the Reserved permission to Unique, and we may weaken t_3 to t_4, since we know that a write access to t_3 yields t_4, and via WEAKEN-VIA-ACCESS *performing an access weakens permissions*. We further elaborate on the nature of *pointwise permission weakening* in section 5.2.

4.2.4 Concurrent Block Separation

$$\texttt{bongo}(y) = \texttt{let}\, x = \texttt{RetagN}(y, 2, \texttt{mut})\ \texttt{in}$$
$$\left(\begin{array}{l} \texttt{let}\, p = \texttt{Retag}(x, \texttt{mut})\ \texttt{in} \\ p \leftarrow 42;\ p \end{array} \;\middle|\middle|\; \begin{array}{l} \texttt{let}\, w = \texttt{Retag}(x +_{\mathrm{L}} 1, \texttt{shared})\ \texttt{in} \\ !\,w;\ w \end{array}\right)$$

(a) Program bongo accepts some reference (or pointer) y, immediately retags, and concurrently writes to the first offset and reads from the second offset. It ends by returning the new reference from each thread in a pair.

BONGO-SPEC

$$\frac{\begin{array}{c} \mathbf{t}_1(\tau) = \mathbf{t}_1' \\ \{z, z+1\} \subseteq \mathsf{dom}(\mathbf{t}_1'.\textsc{data}) \qquad \mathsf{access}(W, \tau, \{z, z+1\}, \mathbf{t}_1) = \mathbf{t}_4 \end{array}}{\begin{array}{l} \{(bid, z) \mapsto\!\!* \ [v_1, v_2] * (bid, z_t) \mapsto\!\!*_t \ \mathbf{t}_1\} \\ \quad \texttt{bongo}((bid, z, \tau)) \\ \left\{\begin{array}{l} w.\ \exists k_x k_p k_w.\ w = ((bid, z, \tau \mathbin{+\!\!+} [k_x, k_p]), (bid, z+1, \tau \mathbin{+\!\!+} [k_x, k_w])) * \\ \tau \mathbin{+\!\!+} [k_x] \notin \mathsf{dom}(\mathbf{t}_1) * (bid, z) \mapsto\!\!* \ [42, v_2] * \\ (bid, z_t) \mapsto\!\!*_t \ \mathbf{t}_4 \left[\tau \mathbin{+\!\!+} [k_x] \leftarrow \mathsf{Tree}\left(\begin{array}{l} \{z \leftarrow \mathsf{Unique}, z+1 \leftarrow \mathsf{Unique}\}, \\ \left\{\begin{array}{l} k_p \leftarrow \mathsf{Tree}(\{z \leftarrow \mathsf{Unique}\}, \emptyset), \\ k_w \leftarrow \mathsf{Tree}(\{z+1 \leftarrow \mathsf{Frozen}\}, \emptyset) \end{array}\right\} \end{array}\right)\right] \end{array}\right\} \end{array}}$$

(b) Iris specification for bongo proved[20] in Rocq.

Figure 4.13: Program bongo in λ^{TB}.

Consider the program bongo shown in fig. 4.13. This program accepts an arbitrary reference y, retags a new node, and then in parallel retags from the new x reference, and writes to the first offset on the left and reads from the second offset on the right. Upon concluding, it returns the new references

[20]Our logic has a caveat that exclusive tags are yet not demonstrable across threads: we are unable to show that $k_p \neq k_w$. See the limitations section.

derived from each thread. The specification should not simply discard the new nodes in the logical tree, since `bongo` returns the new references, and thus clients of `bongo` may access these references.

Consider how we may write a specification for `bongo`. The program performs heap accesses, so its precondition must provide ownership of the values and borrow tree for the accessed offsets. Thus we will need some points-tos $(bid, z) \mapsto\!\!\!* [v_1, v_2]$ and $(bid, z_t) \mapsto\!\!\!* \mathbf{t}_1$ in the precondition. In our proof outline, we implicitly limit the offsets to $\{z, z+1\}$, and break up the precondition and postcondition trees. This is possible with a couple invocations of Split-Grove to obtain "the middle trees".

As for fig. 4.10, we need to know that a *write access* to τ on this arbitrary tree succeeds, since the left thread performs a write access. Furthermore, assuming that the offsets $\{z, z+1\}$ occur in the accessed node at τ in the tree is reasonable, since for the block-spanning ghost trees/groves (recall fig. 4.7) we otherwise cannot constrain which offsets are available throughout the tree, and `bongo` continues to locally access the tree at τ at these offsets.

We may laterally decompose both the set of offsets and then the trees via Lateral-Decomposition, and glue these reasoning principles together with Disjoint-Offsets:

$$\begin{aligned}
\mathsf{access}(acc, \tau, X_1 \cup X_2, \mathbf{t}_1 \cup \mathbf{t}_2) &= \mathsf{access}(acc, \tau, X_1 \cup X_2, \mathbf{t}_1) \cup \mathsf{access}(acc, \tau, X_1 \cup X_2, \mathbf{t}_2) \\
&= \mathsf{access}(acc, \tau, X_2, \mathsf{access}(acc, \tau, X_1, \mathbf{t}_1)) \\
&\quad \cup \mathsf{access}(acc, \tau, X_1, \mathsf{access}(acc, \tau, X_2, \mathbf{t}_2)) \\
&= \mathsf{access}(acc, \tau, X_1, \mathbf{t}_1) \cup \mathsf{access}(acc, \tau, X_2, \mathbf{t}_2)
\end{aligned}$$

Now we provide a proof outline for `bongo`:

$$\left\{\begin{array}{l} \mathsf{access}(W, \tau, \{z, z+1\}, \mathbf{t}_1^l \cup \mathbf{t}_1^r) = \mathbf{t}_4^l \cup \mathbf{t}_4^r * \\ (bid, z) \mapsto\!\!\!* [v_1, v_2] * (bid, z) \mapsto\!\!\!*_t \mathbf{t}_1^l \cup \mathbf{t}_1^r \end{array}\right\}$$

$$\left\{\begin{array}{l} \mathsf{access}(R, \tau, \{z, z+1\}, \mathbf{t}_1^l \cup \mathbf{t}_1^r) = \mathbf{t}_2^l \cup \mathbf{t}_2^r * \\ \mathsf{access}(W, \tau, \{z, z+1\}, \mathbf{t}_2^l \cup \mathbf{t}_2^r) = \mathbf{t}_4^l \cup \mathbf{t}_4^r * \\ (bid, z) \mapsto\!\!\!* [v_1, v_2] * (bid, z) \mapsto\!\!\!*_t \mathbf{t}_1^l \cup \mathbf{t}_1^r \end{array}\right\}$$

$\mathtt{let}\ x = \mathtt{RetagN}(y, 2, \mathtt{mut})\ \mathtt{in}$

$$\left\{\begin{array}{l} \mathsf{access}(W, \tau, \{z, z+1\}, \mathbf{t}_2^l \cup \mathbf{t}_2^r) = \mathbf{t}_4^l \cup \mathbf{t}_4^r * \\ (bid, z) \mapsto\!\!\!* [v_1, v_2] * (bid, z) \mapsto\!\!\!*_t \begin{tikzcd} \mathbf{t}_2^l \cup \mathbf{t}_2^r \arrow[d, dash, dashed, "\tau +\!\!+ [k_x]"] \\ \{z \leftarrow \mathsf{Reserved}, z+1 \leftarrow \mathsf{Reserved}\} \end{tikzcd} \end{array}\right\}$$

$$\left\{\begin{array}{l}
\mathsf{access}(W,\tau,z,\mathbf{t}_2^l) = \mathbf{t}_4^l * \mathsf{access}(W,\tau,z+1,\mathbf{t}_2^r) = \mathbf{t}_4^r * \\
(bid,z) \mapsto v_1 * (bid,z) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_2^l \\ \vdots\ \tau ++ [k_x] \\ \{z \leftarrow \mathsf{Reserved}\}\end{array} * \\
(bid,z+1) \mapsto v_2 * (bid,z+1) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_2^r \\ \vdots\ \tau ++ [k_x] \\ \{z+1 \leftarrow \mathsf{Reserved}\}\end{array}
\end{array}\right\}$$

$$\left(\begin{array}{l}
\left\{\begin{array}{l}
\mathsf{access}(R,\tau,z,\mathbf{t}_2^l) = \mathbf{t}_3^l * \\
\mathsf{access}(W,\tau,z,\mathbf{t}_3^l) = \mathbf{t}_4^l * \\
(bid,z) \mapsto v_1 * \\
(bid,z) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_2^l \\ \vdots\ \tau ++ [k_x] \\ \mathsf{Reserved}\end{array}
\end{array}\right\} \\
\texttt{let}\ p = \texttt{Retag}(x, \texttt{mut})\ \texttt{in} \\
\left\{\begin{array}{l}
\mathsf{access}(W,\tau,z,\mathbf{t}_3^l) = \mathbf{t}_4^l * \\
(bid,z) \mapsto v_1 * \\
(bid,z) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_3^l \\ \vdots\ \tau ++ [k_x] \\ \mathsf{Reserved} \\ k_p \diagup \\ \mathsf{Reserved}\end{array}
\end{array}\right\} \\
p \leftarrow 42;\ p \\
\left\{\begin{array}{l}
(bid,z) \mapsto 42 * \\
(bid,z) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_4^l \\ \vdots\ \tau ++ [k_x] \\ \mathsf{Unique} \\ k_p \diagup \\ \mathsf{Unique}\end{array}
\end{array}\right\}
\end{array}
\Bigg\|
\begin{array}{l}
\left\{\begin{array}{l}
\mathsf{access}(R,\tau,z+1,\mathbf{t}_2^r) = \mathbf{t}_3^r * \\
\mathsf{access}(W,\tau,z+1,\mathbf{t}_3^r) = \mathbf{t}_4^r * \\
(bid,z+1) \mapsto v_2 * \\
(bid,z+1) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_2^r \\ \vdots\ \tau ++ [k_x] \\ \mathsf{Reserved}\end{array}
\end{array}\right\} \\
\texttt{let}\ w = \texttt{Retag}(x +_{\mathrm{L}} 1, \texttt{shared})\ \texttt{in} \\
\left\{\begin{array}{l}
\mathsf{access}(R,\tau,z+1,\mathbf{t}_3^r) = \mathbf{t}_3^r * \\
\mathsf{access}(W,\tau,z+1,\mathbf{t}_3^r) = \mathbf{t}_4^r * \\
(bid,z+1) \mapsto v_2 * \\
(bid,z+1) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_3^r \\ \vdots\ \tau ++ [k_x] \\ \mathsf{Reserved} \\ \diagdown k_w \\ \mathsf{Frozen}\end{array}
\end{array}\right\} \\
!\,w;\ w \\
\left\{\begin{array}{l}
\mathsf{access}(W,\tau,z+1,\mathbf{t}_3^r) = \mathbf{t}_4^r * \\
(bid,z+1) \mapsto v_2 * \\
(bid,z+1) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_3^r \\ \vdots\ \tau ++ [k_x] \\ \mathsf{Reserved} \\ \diagdown k_w \\ \mathsf{Frozen}\end{array}
\end{array}\right\} \\
\left\{\begin{array}{l}
(bid,z+1) \mapsto v_2 * \\
(bid,z+1) \mapsto\!\!\!\ast_t \begin{array}{c}\mathbf{t}_4^r \\ \vdots\ \tau ++ [k_x] \\ \mathsf{Unique} \\ \diagdown k_w \\ \mathsf{Frozen}\end{array}
\end{array}\right\}
\end{array}\right)$$

$$\left\{ (bid, z) \mapsto\!\!\!* [42; v_2] * (bid, z) \mapsto\!\!\!*_t \begin{tikzcd} & \mathbf{t}_4^l \cup \mathbf{t}_4^r \arrow[d, dashed, dash, "\tau \mathbin{+\!\!+} [k_x]"] & \\ & \{z \leftarrow \mathsf{Unique}, z+1 \leftarrow \mathsf{Unique}\} \arrow[dl, dash, "k_p"'] \arrow[dr, dash, "k_w"] & \\ \{z \leftarrow \mathsf{Unique}\} & & \{z+1 \leftarrow \mathsf{Frozen}\} \end{tikzcd} \right\}$$

As shown in fig. 4.13, the precondition requires that a write[21] access on the original tree succeeds, and we use its result in the final tree in the post condition. As illustrated in our proof outline, in order to step through the first `RetagN`, we must first insert a read access before our write access in our hypothesis. Then we may apply WP-RetagN and then prepare for the parallel operator. Via Split-Grove, we partition both the tree, resulting read access, and the final tree resulting from the write access into those with root block domain $\{z_t, \ldots, z\}$ and one with root block domain $\{z+1, \ldots, z_t + |\mathsf{dom}(\mathbf{t}_1.\textsc{data})| - 1\}$. Furthermore, we must partition the write access via Lateral-Decomposition and decompose the offsets accessed, and apply Disjoint-Offsets to remove the vacuous[22] accesses.

Now we may apply WP-Par. In order to produce the right access assumptions, we employ Commute-Insertion after every retag to prepare for the next operation. Furthermore, we may convert our reasoning to per-location ghost trees instead of block-spanning ghost trees/groves via Tree-Accessor[23]. In the left thread, we once again insert a read before the write, then apply WP-RetagN and WP-Store. In the right thread, we insert a read before applying WP-RetagN, and insert another read access before applying WP-Load. Since we need to produce a tree compatible with the write access for the postcondition, we may apply Weaken-Tree to obtain the final tree from the write access assumption, since *trees related by access are related by pointwise permission weakening*[24] by Weaken-via-Access.

[21]It would be possible to prove a more general specification saying that the write access succeeds *just for offset* z, and a read access succeeds *just for offset* $z+1$. To write this specification, we would need to laterally split the pre and post access trees $\mathbf{t}_1$ and $\mathbf{t}_4$ into a tree for the first offset and a tree for the second offset, or in our more general case partition each tree into one with root block domain $\{z_t, \ldots, z\}$ and one with root block domain $\{z+1, \ldots, z_t + |\mathsf{dom}(\mathbf{t}_1.\textsc{data})| - 1\}$. However, for our purposes, such a specification is too cluttered and fails to showcase interesting reasoning principles from fig. 4.12. Furthermore, our provided specification is much more succinct than this more general one, which better demonstrates the flexibility and expressiveness of the logic, since we may verify this program with one access assumption without deconstructing the trees nor accesses in the specification.

[22]The partition with root domain $\{z_t, \ldots, z\}$ is not concerned with $z+1$, and the partition with root domain $\{z+1, \ldots, z_t + |\mathsf{dom}(\mathbf{t}_1.\textsc{data})| - 1\}$ is not concerned with z.

[23]For arbitrary trees, it is not in general possible to convert a grove spanning a single offset into a per-location ghost tree via Tree-to-Grove. This is a consequence of the design of the state interpretation and points-tos, and we address this limitation later.

[24]See section 5.2 for more details.

Upon termination of the threads, we assemble the postcondition via COMBINE-GROVE. This last stage of the proof boils down to proving an equality between the union of trees in the spatial context and the tree in the goal. Here we make ample use of our tree library, performing term rewrites relating unions, insertions, and re-associating insertions. See the full details in the Rocq proof.

Chapter 5

Model

In any Iris-based program logic, one must instantiate Iris's program logic layer by providing the *logical state* for the *state interpretation* (SI). This chapter outlines the ghost state buttressing the user-facing program logic. Each section builds up the ghost state needed for a progressively more expressive program logic, rather than dumping the most intricate version all at once. We adopt the conventions established in section 2.2, and employ diagrams a la fig. 2.4 to illustrate case analysis incumbent upon proving our primitive Hoare logic laws via WP-LIFT-STEP.

5.1 Monolithic Ghost Trees

Outlined in fig. 5.1 are the ghost state connectives for a program logic for Tree Borrows, where ghost trees available to clients of the logic are *monolithic* and reflect the structure of the physical trees exactly. As shown in fig. 5.2, the authoritative map containing the ghost trees may have fewer blocks than the physical state, but for every tree it does have it matches exactly[1] with the physical trees. In the points-to for a ghost tree, one owns the tree for an entire block, hence why it maps from *BlockId*.

This phase establishes the basic architecture for the whole development. The ghost state for block sizes and values as shown in fig. 5.1 remained unchanged for the rest of the development. The points-to connectives for values and trees must also own knowledge of the block's size, and block sizes *never change*, thus a persistent block size achieves these ends. Concretely, the points-to for block sizes enables users to infer how much of the array of values they own, and at this stage one must own the entire tree for the whole block. Thus the importance of a persistent points-to for block sizes as

[1] In the Rocq implementation, the ghost trees were exactly equal to a transformation on the physical tree that converted every vector in the physical tree to a gmap with the same data. We denote this equality as $\equiv_t$ within fig. 5.2.

$$\begin{aligned}
S_s(\sigma) &\triangleq \exists m_s : \mathit{BlockId} \overset{\text{fin}}{\rightharpoonup} \mathbb{N}.\ \boxed{\bullet(\text{ghost\_map}(m_s))}^{\gamma_s} * m_s \preceq_s \sigma \\
\mathit{bid} \mapsto_s n &\triangleq \boxed{\mathit{bid} \overset{\square}{\hookrightarrow} n}^{\gamma_s} * 0 < n \\
S_v(\sigma) &\triangleq \exists m_v : \mathit{Loc} \overset{\text{fin}}{\rightharpoonup} \mathit{Val}.\ \boxed{\bullet\text{ghost\_map}(m_v)}^{\gamma_v} * m_v \preceq_v \sigma \\
\ell \overset{dq}{\mapsto} v &\triangleq \boxed{\ell \overset{dq}{\hookrightarrow} v}^{\gamma_v} * \exists n.\ \ell.\textsc{block\_id} \mapsto_s n * 0 \leq \ell.\textsc{offset} < n \\
\ell \overset{dq}{\mapsto\!\!*}\ \mathbf{v} &\triangleq \left(\mathop{\huge *}_{i \mapsto v \in \mathbf{v}} \boxed{\ell +_{\mathrm{L}} i \overset{dq}{\hookrightarrow} v}^{\gamma_v} \right) * 0 \leq \ell.\textsc{offset} \leq \ell.\textsc{offset} + |\mathbf{v}| \leq n \\
S_t(\sigma) &\triangleq \exists m_t : \mathit{BlockId} \overset{\text{fin}}{\rightharpoonup} \mathit{Tree}(\mathbb{N}, \mathit{Protection} \times (\mathbb{N} \overset{\text{fin}}{\rightharpoonup} \mathit{LocState})). \\
&\qquad \boxed{\bullet(\text{ghost\_map}(m_t))}^{\gamma_t} * m_t \preceq_t \sigma \\
\mathit{bid} \mapsto\!\!*_t\ \mathbf{t} &\triangleq \boxed{\mathit{bid} \hookrightarrow \mathbf{t}}^{\gamma_t} * \exists n.\ \mathit{bid} \mapsto_s n * \text{every } \mathbf{t} \text{ node spans } [0, n) \\
S(\sigma) &\triangleq S_s(\sigma) * S_v(\sigma) * S_t(\sigma)
\end{aligned}$$

Figure 5.1: Ghost state for monolithic logical trees/groves. One owns a ghost tree for the entire block, and every block at every node has the full block domain.

$$m_t \preceq_t \sigma \triangleq \forall \mathit{bid}\ \mathbf{t}.\ m_t(\mathit{bid}) = \mathbf{t} \rightarrow \exists \mathit{blk}.\ \sigma(\mathit{bid}) = \mathit{blk} \wedge \mathbf{t} \equiv_t \mathit{blk}.\textsc{tree}$$

Figure 5.2: Relation between monolithic ghost trees/groves and physical trees. Note that $\mathbf{t} \equiv_t \mathit{blk}.\textsc{tree}$ entails that the trees have the same structure and content.

first shown in fig. 4.9 becomes clear. Optional blocks $\mathit{Block}^?$ in the physical heap (fig. 4.4) help to ensure that block size points-tos can be persistent, since persistence means that these points-tos are not "consumed" by WP-Free.

In separation logics for languages with block-based memory, a points-to for block sizes is important for `Free`, since one typically needs to own the resources for the entire block in order to free the block, as shown in fig. 4.8. However, in this first model, we also require block size ownership for all of the other primitive laws. The use of block size points-tos in the other primitive laws is unusual, and we rectify this in section 5.4.

With this set up, the model is sound and "reasonably complete": it is able to reason about straight, sequential programs, such as `main` in fig. 4.6. It is also able to reason about arbitrary trees. Unfortunately, it cannot yet reason about the programs nor specifications from fig. 4.10 nor fig. 4.13.

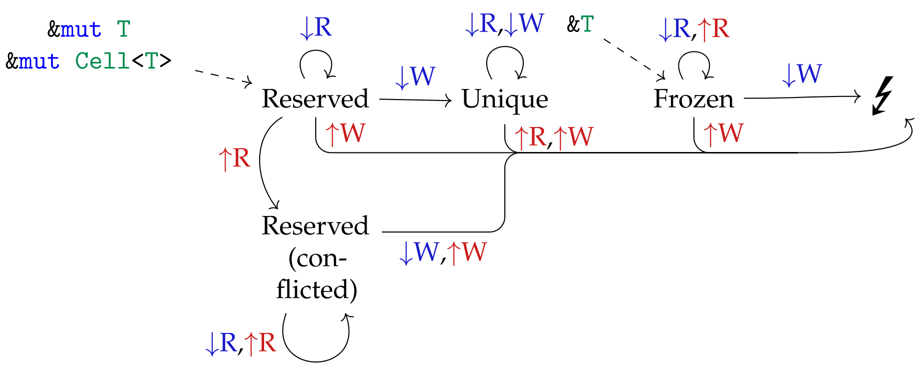


Figure 5.3: Diagram of the protected permission state machine from Villani et al. [31]. UB triggers upon reaching ↯. We label transitions by the events that cause them: (R)ead or (W)rite, each either ↑(foreign) or ↓(local).

5.2 Pointwise Permission Weakening

In over-approximate reasoning, it is critical to have some way to weaken knowledge about the state, otherwise one is essentially executing the dynamic semantics in all of its gratuitous detail. In a program logic for Tree Borrows, knowing the exact tree in the execution quickly becomes irrelevant. Moreover, the ability to reconcile potentially different permissions originating from different execution branches curbs proof state explosion and spurious complexity. Thus possessing some means to obtain a *strongest possible weakening* between Tree Borrows permissions is crucial to the utility and scalability of a Tree Borrows program logic.

Thus this section introduces the machinery needed to achieve *pointwise permission weakening*, which we used to prove the specification for `foo` in fig. 4.10. Consider again the state machine for unprotected Tree Borrows permissions as shown in fig. 2.2. A "weakening" relation should respect this state machine. However, the state machine shown in fig. 2.2 is merely an approximation of the real Tree Borrows state machine, which does not easily lend itself to a nice, clean diagram.

Tree Borrows also has a notion of *protected*[2] permissions, which have a modified version of the state machine, as shown in fig. 5.3. As described in more detail in Villani et al. [31], *protectors* ensure that some nodes *do not become* Disabled. This guarantees that a particular reference corresponding to that protected node stays alive. Protectors are specifically attached to

[2]Our core calculus does not support protected retags, and we leave full protector support to future work. However, we believe it important to not entirely ignore protectors in order to aid future development. Thus our Rocq implementation includes the full Tree Borrows state machine for protectors, and protectors are not ignored in the logical state. While significant work remains to fully incorporate protectors into the program logic, the foundation already laid and machinery developed in these sections will lessen and assuage future refactoring.

references at function call boundaries, which ensures that Tree Borrows is consistent with the lifetime enforcement of the borrow checker, guaranteeing that the reference's lifetime persists at least until the end of the function. Tree Borrows protectors are the Rust analogue to *noalias* attached to function arguments in LLVM, which LLVM uses to guarantee that no other pointers outside of a function call modify the location during the function's execution.

Since the protected state machine has different behavior than that of unprotected nodes, we require distinct weakening relations for unprotected and protected permissions. "Staying alive" is basically equivalent to "not becoming Disabled", and a node generally only transitions to Disabled under a *foreign write* ($\uparrow$W). These transitions are important since they ensure that accesses to references are *well-bracketed*, and end the lifetimes of references in other parts of the borrow tree. When we place a node under protection, we effectively forbid foreign writes to this node, restricting how other parts of the code can interact with this location.

The diagrams in fig. 2.2 and fig. 5.3 conceal that the full Tree Borrows state machine tracks whether or not a location was previously accessed. Recall fig. 4.4. The accessed bits are part of the physical state, and with the Tree Borrows permissions forms the *location states*, which comprise the true states of the Tree Borrows state machines[3]. The accessed bit plays a key role in the protected state machine. Specifically, foreign writes to a protected node only lead to UB if the accessed bit is true, if an access as already occurred to this reference. Essentially, we only bother to forbid foreign writes if some other part of the code accesses this reference. This further complicates the design of a weakening relation, since in general we require a weakening relation between location states, not just the naked permission.

As a first attempt at a weakening relation, we essentially use reachability in the state machine for this relation. We say that a location state ls_1 is "weaker" than another location state ls_2 if ls_1 is reachable from ls_2 in the state machine. It is fundamental to our program logic that location states connected by a transition in the state machine satisfy our weakening relation. In other words, if $\mathsf{access}_{prot}(acc, locality, ls_2) = ls_1$ holds in the state machine, then we must have $ls_1 \preceq_{prot} ls_2$. This is an important property to users of the program logic, which we denote as Weaken-via-Access. For instance, consider if in one branch the code performs a local read, and in another the code performs a local write, as in fig. 4.10. Let the starting permission be Reserved. In the read branch, the permission will remain as Reserved. However, the write branch will update the permission to Unique. In the postcondition of Foo-Spec, we would like to own the strongest possible borrow tree that respects both branches, and in this case the reference in question should have a Unique permission. Thus, the read branch needs to be able to weaken the reference's

[3]See the Rocq development for the full Tree Borrows state machine in all of its glory.

permission from Reserved to Unique. Defining $ls_1 \preceq_{prot} ls_2$ as reachability satisfies this property.

$$\begin{tikzcd}
ls_1 \arrow[r, "\preceq"] \arrow[d, "\text{access}"] & ls_1^{phys} \arrow[d, dashed, "\text{access}"] & \mathbf{n}_1 \arrow[r, "\preceq"] \arrow[d, "\text{access}"] & n_1^{phys} \arrow[d, dashed, "\text{access}"] & \mathbf{t}_1 \arrow[r, "\preceq"] \arrow[d, "\text{access}"] & t_1^{phys} \arrow[d, dashed, "\text{access}"] \\
ls_2 \arrow[r, dashed, "\preceq"] & ls_2^{phys} & \mathbf{n}_2 \arrow[r, dashed, "\preceq"] & n_2^{phys} & \mathbf{t}_2 \arrow[r, dashed, "\preceq"] & t_2^{phys}
\end{tikzcd}$$

Figure 5.4: Diagram representing the safety and preservation proofs needed for WP-LIFT-STEP. We must lift safety and preservation from location states to nodes and trees. The solid arrows indicate premises of the simulation property, and the dashed arrows indicate the conclusion.

However, there is another important class of property required of the weakening relation in order to prove the primitive laws of the program logic via WP-LIFT-STEP. Recall from section 2.2.3, that in order to prove the primitive laws via WP-LIFT-STEP, we must prove *safety* and *preservation*. In order to prove safety and preservation for our Tree Borrows program logic, the weakening relation needs to be a *simulation* of the Tree Borrows state transition system, as shown in fig. 5.4.

Let us consider more precisely the proof goal for preservation. That is, given hypotheses $\text{access}_{prot}(acc, locality, ls_1) = ls_2$, $\text{access}_{prot}(acc, locality, ls_1^{phys}) = ls_2^{phys}$ and $ls_1 \preceq_{prot} ls_1^{phys}$, we need to be able to show that $ls_2 \preceq_{prot} ls_2^{phys}$. Unfortunately, reachability does not satisfy preservation in the unprotected state machine.

To demonstrate this, let us consider a counterexample for preservation. Let $ls_1 = (\texttt{true}, \text{Reserved})$, $ls_1^{phys} = (\texttt{false}, \text{Reserved})$, $ls_2 = (\texttt{true}, \text{Disabled})$, and $ls_2^{phys} = (\texttt{false}, \text{Disabled})$. Let our premises include:

- $\text{access}_{\text{Unprotected}}(\uparrow W, (\texttt{true}, \text{Reserved})) = (\texttt{true}, \text{Disabled})$, a transition in fig. 2.2.
- $\text{access}_{\text{Unprotected}}(\uparrow W, (\texttt{false}, \text{Reserved})) = (\texttt{false}, \text{Disabled})$, a transition in fig. 2.2.
- $(\texttt{true}, \text{Reserved}) \preceq_{\text{Unprotected}} (\texttt{false}, \text{Reserved})$, which in the case of reachability is satisfied by the transition $\text{access}_{\text{Unprotected}}(\downarrow R, (\texttt{false}, \text{Reserved})) = (\texttt{true}, \text{Reserved})$, since a local access always sets the accessed bit to `true`.

Thus, to establish preservation in this case, we would need to show $(\texttt{true}, \text{Disabled}) \preceq_{\text{Unprotected}} (\texttt{false}, \text{Disabled})$, and in the case of reachability we need to show that there is a sequence of transitions from $(\texttt{false}, \text{Disabled})$ to $(\texttt{true}, \text{Disabled})$ in the unprotected state machine. Unfortunately, no such sequence of transitions exists, because a local access to a disbaled permission incurs UB. Thus, reachability does not satisfy preservation for the unprotected state machine.

$$(\_, \mathsf{Disabled}) \preceq (\_, \mathsf{Frozen}) \preceq (\_, \mathsf{Unique}) \preceq (\_, \mathsf{Reserved})$$

(a) Unprotected location state weakening. Neither the accessed bit nor conflictedness matter.

$$\begin{tikzcd}
 & & (\bot, R) \arrow[dl] \arrow[dr] & & & & \\
 & (\top, R) \arrow[dl] \arrow[dr] & & (\bot, R^{☣}) \arrow[dl] \arrow[dr] & & (\bot, F) \arrow[dl] \arrow[dr] & \\
(\top, U) & & (\top, R^{☣}) & & (\bot, D) & & (\top, F)
\end{tikzcd}$$

(b) We chart protected location state weakening in a *Hasse Diagram*. We denote a `true`, accessed bit as $\top$, and a `false`, unaccessed bit as $\bot$. Furthermore, we denote Reserved as R, Reserved conflicted as $R^{☣}$, Unique as U, Frozen as F, and Disabled as D. The partial ordering goes from the top of the diagram for the strongest location states to the bottom of the diagram for the weakest location states.

Figure 5.5: Location state weakening.

Thus, for unprotected nodes, we essentially use reachability *without considering the accessed bit*, shown in fig. 5.5a. The unprotected relation also holds between any two Reserved permissions, conflicted or otherwise (harmonious). It turns out that reachability still works for protected nodes, thus we employ it for that case. We diagram the full weakening relation for protected location states in fig. 5.5b.

$$\begin{aligned}
m_t \preceq_t \sigma &\triangleq \forall bid\ \mathbf{t}.\ m_t(bid) = \mathbf{t} \to \exists blk.\ \sigma(bid) = blk \land \mathbf{t} \preceq blk.\textsc{tree} \\
S_t(\sigma) &\triangleq \exists m_t : BlockId \xrightharpoonup{\text{fin}} Tree(\mathbb{N}, Protection \times (\mathbb{N} \xrightharpoonup{\text{fin}} LocState)). \\
&\qquad \boxed{\bullet(\text{ghost\_map}(m_t))}^{\gamma_t} * m_t \preceq_t \sigma \\
bid \mapsto\!\!*_t\ \mathbf{t} &\triangleq \exists \mathbf{t}'.\boxed{bid \hookrightarrow \mathbf{t}'}^{\gamma_t} * \mathbf{t} \preceq \mathbf{t}' * \\
&\qquad \exists n. bid \mapsto_s n * \text{every } \mathbf{t}' \text{ node spans } [0, n)
\end{aligned}$$

Figure 5.6: Ghost state supporting pointwise permission weakening. This features the relation between ghost trees and physical trees, where we lift pointwise permission weakening to the tree level, and both trees have the same structure and offsets. One owns a ghost tree for the entire block, and every block at every node has the full block domain.

In fig. 5.6, we present the updated relation used in the SI, where the weakening relation for permissions is pointwise lifted over the tree structure, and the ghost tree still has the same structure as the physical tree. The SI and points-tos now employ the weakening relation as shown in fig. 5.5. The ghost tree points-to now additionally existentially quantifies another ghost tree $\mathbf{t}'$, which is the actual owned tree, and the user-visible ghost tree $\mathbf{t}$ is pointwise weaker than the owned $\mathbf{t}'$. By existentially quantifying over the owned ghost tree, this allows clients of the program logic to weaken the trees in the points-to *without needing direct access to the SI*. If the user-visible ghost tree was the owned tree in the points-to, then weakening the tree would require invoking a ghost state update lemma using the authoritative ghost map in the SI, which would require exposing the SI to clients. Typically, we only want the SI directly accessed up to the layer proving the primitive laws of the Hoare logic. Alternatively, one could restrict weakening to require a weakest precondition predicate. Technically, this does "hide" the SI from clients, but it still requires ownership of the SI in order to logically weaken a tree. By concealing the owned ghost tree behind an existential quantifier and taking advantage of reflexivity and transitivity of our location state weakening relation, users can replace a ghost tree in a points-to with a pointwise weaker tree without owning the SI.

In reproving the primitive laws, we now need to utilize safety and preservation for location states, and lift it pointwise over the entire tree structure as shown in fig. 5.4. The primitive laws as shown in fig. 4.8 require the premise that the access on the ghost tree succeeds and produces and new ghost tree. In proving the safety case of the weakest precondition via WP-LIFT-STEP, we need to produce a new physical tree that the operational semantics can step to, and in the preservation case we need this new physical tree to satisfy the weakening relation for the new ghost tree.

5.3 Subtree Deletion

Executing any sufficiently intricate program under Tree Borrows may incur unbridled growth in the borrow trees, especially as references get passed down the call stack. After a while, most of the references deeper in the borrow tree become ignored by and irrelevant to the rest of the program, or even expire (become Disabled), and thus can no longer be locally accessed, essentially rendered dead weight. In its Tree Borrows implementation, Miri analyzes the entire program to determine which nodes become irrelevant and may be deleted. Suppressing and concealing this growth from clients of the program logic is crucial to its usability and scalability. Therefore, we introduce another important logical weakening: the ability to prune subtrees from the logical tree. The requires modifying the weakening relation between ghost trees and physical trees in the SI, as well as between ghost trees in the points-to to allow the left-hand-side (LHS) trees to have a *prefix of the structure* of the right-hand-side (RHS) trees. In other words, the address space of the LHS tree may be a subset of the address space of the RHS tree. However, it is not sufficient to *just* allow this structural weakening when considering protected nodes.

In order to illustrate the issue with merely employing naive structural weakening, consider reproving safety from fig. 5.4. Introducing naive structural weakening sabotages safety in the protected case. To prove safety, we know that $\mathbf{t}_1 \preceq t_1^{phys}$, and we must show that there exists some t_2^{phys} such that $\mathsf{access}(acc, \tau, t_1^{phys}) = t_2^{phys}$. Consider an accessed and protected permission within a node only in the physical tree. Recall the protected state machine from fig. 5.3. Any foreign write $\uparrow$W to this location will incur UB, and therefore must be prevented. However, since this node only occurs in the physical tree, there is nothing in the premises to prevent this behavior.

Thus we need to refine our notion of structural weakening with a predicate on RHS-only nodes. This predicate requires that *protected nodes are and remain unaccessed*. Let us refer to this predicate as Ψ_{RHS}. In the safety proof, since we require protected RHS-only nodes in the physical tree to be unaccessed, we may conclude that the foreign access to these nodes will succeed. Our refined structural and pointwise weakening relation is now sufficient to prove safety and preservation, but introduces more case analysis into safety and preservation proofs.

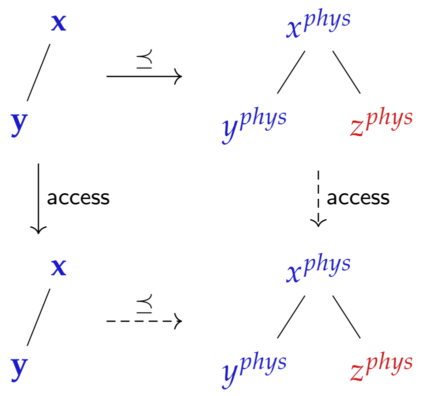


Figure 5.7: Safety and preservation proof under *subtree deletion*. We illustrate the proof for an access to node y. This results in local accesses for nodes x and y, notated in blue, and a foreign access to z, notated in red. The trees on the LHS represent *ghost trees* visible to users of the program logic, while the trees on the RHS represent *physical trees* owned by the underlying SI. For safety, we must exploit $\Psi_{\mathsf{RHS}}(z^{phys})$ in order to show an access will succeed. For preservation, we must ensure that Ψ_{RHS} holds for z^{phys} in the bottom-right tree.

In particular, the preservation proof in the RHS-only case must be sensitive to the relative locality of the current address with respect to the access address. Regard fig. 5.7, which illustrates an access to node y. We must ensure that the access is foreign with respect to z, since some local accesses would result in UB and would violate Ψ_{RHS} by flipping on the accessed bit. Fortunately, this is possible to demonstrate, since we restrict accesses to user-facing ghost trees in the points-tos to addresses within that tree's domain, as shown in the primitive laws in fig. 4.8. For fig. 5.7, the user-facing program logic may only directly access nodes x and y, but not z since z is not even present in the LHS logical tree. This ensures that any access to a RHS-only node such as z is foreign.

Furthermore, this refinement complicates the proof of transitivity of the overall weakening relation for trees. Recall from the from previous section that transitivity is a key property ensuring the feasibility of the existentially quantified ghost tree construction of the points-to. In the proof for transitivity of the weakening relation, we must show that $\mathbf{t}_1 \preceq \mathbf{t}_3$, given that $\mathbf{t}_1 \preceq \mathbf{t}_2$ and $\mathbf{t}_2 \preceq \mathbf{t}_3$. Consider the case where there are some nodes $\mathbf{n}_2$ and $\mathbf{n}_3$ and some tag τ such that $\mathbf{t}_2(\tau).\textsc{data} = \mathbf{n}_2$ and $\mathbf{t}_3(\tau).\textsc{data} = \mathbf{n}_3$, but $\tau \notin \mathsf{dom}(\mathbf{t}_1)$. In this case, we know that $\Psi_{\mathsf{RHS}}(\mathbf{n}_2)$ and $\mathbf{n}_2 \preceq \mathbf{n}_3$, and we must show that $\Psi_{\mathsf{RHS}}(\mathbf{n}_3)$. Fortunately, this property between weakening and the RHS predicate holds, and this is a result of the behavior of the Tree Borrows state machines. More precisely, this relies upon the observation that since $\mathbf{n}_2 \preceq \mathbf{n}_3$ holds, we may conclude that the accessed bits in $\mathbf{n}_3$ may only be `true` if they are also `true` in $\mathbf{n}_2$, which follows from fig. 5.5b. But since we know that all of the accessed bits in $\mathbf{n}_2$ are `false`, we may conclude that the accessed bits must also be `false` in $\mathbf{n}_3$.

5.4 Lateral Ghost Tree Separation

So far, the ghost state does not allow users to break up ghost trees across the block. The ability to break up resources across *individual memory locations* is crucial for any concurrent separation logic, and in separation logics for languages with block-based memory, this necessitates the ability to break apart *memory blocks* and their resources. For instance, if one would like to slice an array asunder and give each half to concurrent threads, then the separation logic must provide a means to break apart the resources for the block. Thus, we now provide the infrastructure to separate ghost trees *laterally*.

At first glance, lateral tree separation does not appear particularly threatening. Consider some ghost tree for an entire block that we would like to divide in twain, and pass each half into different threads. These new trees have the same address space and structure as the former tree, but each node contains

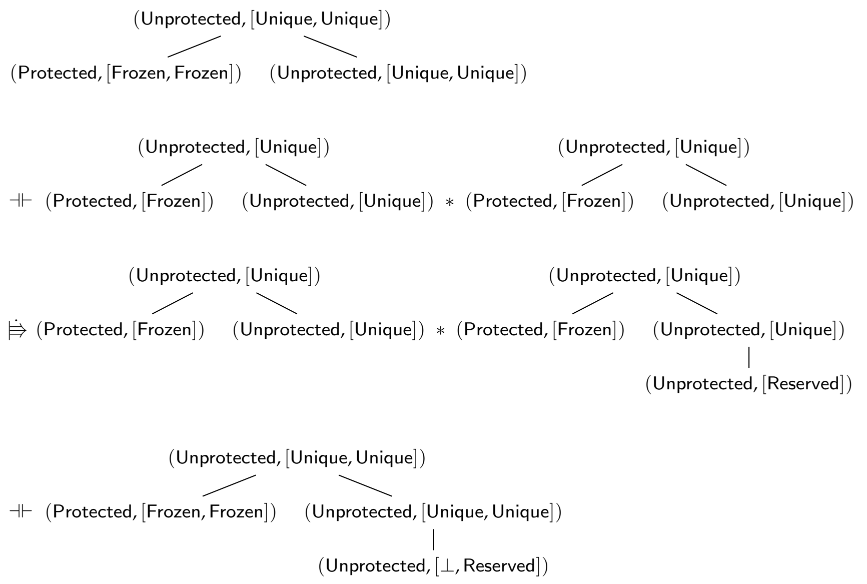


Figure 5.8: Lateral separation and recombination of ghost trees. After separating the trees, we retag on the right tree, and then recombine.

complementary permissions for disjoint offsets of the block. However, each thread may retag their ghost tree, and if we wish to recombine, we need to reconcile these trees with different structures.

Consider the diagram in fig. 5.8. We may split our original ghost tree, which contains two offsets at each node, into one half with only the first offset, and another with just the second offset. Then we retag the right tree with the second offset, and then recombine. Now, we have a ghost tree where different nodes may have different subsets of the block's domain. After recombining, the new node still just has the second offset. In general, we must support ghost trees where subtrees may have subsets of the root node's block domain. Later, we will illustrate exactly what it means for a ghost tree to be *well-formed*, since we cannot allow nodes to have entirely arbitrary subsets of the block's domain.

Permitting different nodes to have different subsets of the block's domain may seem unattractive, as it incurs many more degrees of freedom in the ghost trees. In fig. 5.8 for instance, it may be tempting to recover the permission for the first offset of the new node. However, it is unclear how to recover the Reserved permission from the physical tree, nor how to constrain such permissions in the physical tree. It may be possible to just make this missing permission Disabled, but this likely requires a ghost state update with the SI, which in general we would like to conceal from users. Thus we allow some degrees of freedom for the block subsets, and we later illustrate the precise notion of well-formed ghost trees.

Once we allow lateral separation, we also need to ensure that protectors agree across the blocks for every node. Consider again fig. 5.8. Recall that in Tree Borrows, protectors apply for the entire block, not per-location. When combining trees laterally, we must be able to reconcile the values for the protectors at every node. Disagreeing protector values for the same node must incur a contradiction.

$$
\begin{aligned}
S_p(\sigma) &\triangleq \exists m_p : (\mathit{BlockId} \times \mathit{Tags}) \xrightharpoonup{\text{fin}} \mathit{Protection}. \\
&\quad \boxed{\bullet(\mathsf{ghost\_map}(m_p))}^{\gamma_p} * m_p \preceq_p \sigma * \\
&\quad \mathop{\Huge *}_{bid \mapsto blk \in \sigma} \left(\mathop{\Huge *}_{\tau \mapsto prot \in (\textsc{prot} \mathbin{<\$>} blk.\textsc{tree})} \boxed{(bid, \tau) \overset{\square}{\hookrightarrow} prot}^{\gamma_p} \right) \\
bid \mapsto_p pt &\triangleq \mathop{\Huge *}_{\tau \mapsto prot \in pt} \boxed{(bid, \tau) \overset{\square}{\hookrightarrow} prot}^{\gamma_p} \\
S_t(\sigma) &\triangleq \exists m_t : \mathit{Loc} \xrightharpoonup{\text{fin}} \mathit{Tree}(\mathbb{N}, \mathit{Protection} \times \mathit{LocState}). \\
&\quad \boxed{\bullet(\mathsf{ghost\_map}(m_t))}^{\gamma_t} * m_t \preceq_t \sigma \\
\ell \mapsto_t t &\triangleq \exists t'. \boxed{\ell \hookrightarrow t'}^{\gamma_t} * t \preceq t' * \ell.\textsc{block\_id} \mapsto (\textsc{prot} \mathbin{<\$>} t') * \\
&\quad \exists n.\ \ell.\textsc{block\_id} \mapsto_s n * 0 \le \ell.\textsc{offset} < n \\
\ell \mapsto\!\!*_t \mathbf{t} &\triangleq \exists \mathbf{t}'. \left(\mathop{\Huge *}_{i \mapsto t' \in \mathbf{t}'} \boxed{(\ell.\textsc{block\_id}, i) \hookrightarrow t'}^{\gamma_t} \right) * \\
&\quad \ell.\textsc{block\_id} \mapsto (\textsc{prot} \mathbin{<\$>} \mathbf{t}') * \\
&\quad \mathbf{t} \preceq \mathbf{t}' * \mathsf{WF}(\mathbf{t}) * \mathsf{WF}(\mathbf{t}') * \mathsf{dom}(\mathbf{t}.\textsc{data}) = \mathsf{dom}(\mathbf{t}'.\textsc{data}) * \\
&\quad \exists n.\ \ell.\textsc{block\_id} \mapsto_s n * \\
&\quad \mathsf{dom}(\mathbf{t}.\textsc{data}) = \{\ell.\textsc{offset}, \ldots, \ell.\textsc{offset} + |\mathsf{dom}(\mathbf{t}.\textsc{data})| - 1\} * \\
&\quad \mathsf{dom}(\mathbf{t}.\textsc{data}) \subseteq \{0, \ldots, n-1\} \\
S(\sigma) &\triangleq S_s(\sigma) * S_v(\sigma) * S_t(\sigma) * S_p(\sigma)
\end{aligned}
$$

Figure 5.9: Ghost state supporting lateral tree separation.

Consequently, as shown in fig. 5.9, in order to support lateral tree separation, we must refactor the ghost state and introduce new ghost state for protectors. No longer does the ghost map for trees m_t map block identifiers to ghost trees spanning the whole block. Henceforth, the tree ghost map maps *locations* to *per-location ghost trees*. Thus the primitive logical connective for ghost trees is a points-to $\ell \mapsto_t t$. Recall the primitive laws from fig. 4.8. Most language primitives for heap operations, such as load, store, and atomic reads and writes, only operate on a single memory location, rather than a span of the block. Therefore, this $\ell \mapsto_t t$ is useful for single-location operations and accesses. As we will discuss later, operations on block spanning ghost trees require more assumptions, which is why we believe it prudent to give users access to the single location ghost trees. The systems-programming minded reader may point out that in real programming languages, hardware organizes the memory layout of values across multiple bytes, and that a single load or store from memory often reads from multiple locations since different values require different memory sizes. Our model is somewhat unrealistic, since it may store any arbitrary value at a single location in memory, no matter how large or small. Under a model that more precisely reflects the realities of memory and bytes, it would be sensible for the program logic to perhaps provide some notion of a "constant size ghost tree", where every node in the ghost tree has the same size and block offsets. Our implementation does not make an attempt to hide the underlying memory organization, so we leave the exploration of such alternatives to future work.

However, it is still sometimes useful to reason about ghost trees that span the block as well, hence we also provide a more general $\ell \mapsto\!\!*_t \mathbf{t}$ connective. This is particularly useful for retagging, where the client sees a single insertion into a single block spanning tree, rather than multiple insertions over a list of per-location trees. As shown in fig. 5.9, we define this connective as a separating conjunction over multiple per-location ghost trees spanning its range of the block. In Rocq, we compute a gmap of per-location ghost trees from the block spanning ghost trees. For this computation to work properly, we need to assume that the tree is well-formed, written as $\mathsf{WF}(\mathbf{t}')$.

5.4.1 Well-formed Ghost Trees

$$\begin{aligned}\mathsf{WF}(\mathbf{t}) \triangleq\ & \forall ks_1\ ks_2\ \mathbf{n}_1\ \mathbf{n}_2.\ \mathbf{t}(ks_1).\textsc{data} = \mathbf{n}_1 \to \mathbf{t}(ks_1 \mathbin{+\!\!+} ks_2).\textsc{data} = \mathbf{n}_2 \to \\ & \mathsf{dom}(\mathbf{n}_2) \subseteq \mathsf{dom}(\mathbf{n}_1)\end{aligned}$$

Figure 5.10: The notion of well-formed block spanning ghost trees.

Here we consider what it concretely means for a block spanning ghost tree to properly represent a collection of contiguous per-location ghost trees. Furthermore, here we will illustrate how block spanning ghost trees may have varying subblocks at each node, and how this relates to subtree deletion.

Recall from section 5.3 that the ghost trees may contain a substructure of the physical tree, and this becomes further reduced by the weakening relation between the user-facing ghost tree and the hidden, existentially quantified ghost tree in the points-to. We still support this structural weakening, but now also on the granularity of individual memory locations, not the entire block. As a result, different per-location trees from the same block may have different structures.

$$\begin{tikzcd}[column sep=small]
{[q_0,q_1,q_2,q_3,q_4,q_5]} \arrow[d, dash] & \dashv\vdash & q_0 \arrow[d, dash] & * & q_1 \arrow[d, dash] & * & q_2 \arrow[d, dash] & * & q_3 \arrow[d, dash] & * & q_4 & * & q_5 \arrow[d, dash] \\
{[r_0,r_1,r_2,r_3,\bot,r_5]} \arrow[d, dash] & & r_0 & & r_1 \arrow[d, dash] & & r_2 \arrow[d, dash] & & r_3 \arrow[d, dash] & & & & r_5 \arrow[d, dash] \\
{[\bot,s_1,s_2,s_3,\bot,s_5]} \arrow[d, dash] & & & & s_1 \arrow[d, dash] & & s_2 & & s_3 \arrow[d, dash] & & & & s_5 \arrow[d, dash] \\
{[\bot,t_1,\bot,t_3,\bot,t_5]} \arrow[d, dash] & & & & t_1 & & & & t_3 \arrow[d, dash] & & & & t_5 \arrow[d, dash] \\
{[\bot,\bot,\bot,u_3,\bot,u_5]} \arrow[d, dash] & & & & & & & & u_3 \arrow[d, dash] & & & & u_5 \\
{[\bot,\bot,\bot,v_3,\bot,\bot]} & & & & & & & & v_3
\end{tikzcd}$$

Figure 5.11: Visualization of well-formedness. This represents a slice of the trees along a single address path $[q,r,s,t,u,v]$. The LHS tree slice is from a *block-spanning tree* for block offsets $\{0,\ldots,5\}$, and the RHS is slices of the corresponding *per-location trees* also for block offsets $\{0,\ldots,5\}$. This provides justification for the definition of well-formedness.

Consider a single path down the address space for some contiguous collection of per-location ghost trees. We essentially have a list of lists of varying lengths, all "hanging" from the same root. This is the natural structure a common address path exhibits for our per-location trees, the ghost trees actually owned in the ghost map, shown in the RHS of fig. 5.11, exhibiting slices of the per-location trees for offsets $\{0,\ldots,5\}$ along address path $[q,r,s,t,u,v]$. Now consider how this list of lists looks if we combine all of these locations into a single span of the block, shown in the LHS of fig. 5.11. Since all lists "hang" from the root, we have a full, continuous span of the block at the root of the new, block spanning tree. But since the lists may have different lengths (some subtrees in the per-location trees may have been deleted or not retagged at some addresses but not for others), the new nodes down the path for the block spanning ghost tree will have fewer offsets from the root block span. The set of offsets in a node below the root may not even be continuous, since the middle per-location ghost tree could be missing that tag in its address space. We exhibit such structure in fig. 5.11, where the LHS tree misses entries for offsets $\{0,2,4\}$ for the t node, since this address is missing from the per-location trees at these same offsets.

Moreover, notice that the block subdomains of the new block spanning ghost nodes along a path down the address space are not random, but *parent nodes have a superset of block offsets of each of their children*. Consider again the image of lists or "strings" hanging from some common height, representing a common path down the address space. Examine again the tree slices in fig. 5.11. These strings may have different lengths, as shown for the RHS tree slices in fig. 5.11. At each progressively lower height, the set of strings still present at that height is a subset of the previous height, and so on. This physical intuition hints at a justification the formal property we require for our block-spanning ghost trees: as exhibited by the LHS tree slice in fig. 5.11, along the path $[q, r, s, t, u, v]$ the tree's nodes feature offsets $\{0, 1, 2, 3, 4, 5\} \supseteq \{0, 1, 2, 3, 5\} \supseteq \{1, 2, 3, 5\} \supseteq \{1, 3, 5\} \supseteq \{3, 5\} \supseteq \{3\}$. Along every path down the tree, the child nodes have a subset of the block domain of their parent nodes. We capture this well-formedness property formally in fig. 5.10.

Since we now describe our block-spanning ghost trees $\mathbf{t}$ by $\mathsf{WF}(\mathbf{t})$, we need to further refine our weakening relation. From section 5.3, recall the property Ψ_{RHS}: every protected node only on the right-hand side must be unaccessed. This prevents foreign writes from causing UB. The property Ψ_{RHS} applies *per-node*, since previously weakening could only capture structural differences *at the granularity of entire nodes*. In order to facilitate lateral separation, we must be able to capture structural differences *at the granularity of individual locations*. For block-spanning ghost nodes $\mathbf{n}_1$ and $\mathbf{n}_2$, $\mathbf{n}_1 \preceq \mathbf{n}_2$ now entails that $\mathsf{dom}(\mathbf{n}_1) \subseteq \mathsf{dom}(\mathbf{n}_2)$. Therefore, we require a refined predicate ψ_{RHS} over *individual location states* applying to location states at offsets in $\mathbf{n}_2$ but absent from $\mathbf{n}_1$. Analogously to Ψ_{RHS} for nodes, $\psi_{RHS}(ls)$ requires that ls either inhabits an unprotected node, or if it inhabits a protected node that it is unaccessed.

The well-formedness property is fundamental to reasoning about ghost trees under lateral separation. Consider a case of preservation of WP-LIFT-STEP, shown in fig. 5.12, where the LHS user-facing ghost tree is directly accessed at node y at offset 1. We need the preservation property to remain valid for block-spanning ghost trees, as in the case of WP-RetagN, where one performs a read access across the retag range. Before we introduced lateral separation, any access to any offset in the block was implicitly within the range of the ghost tree's block span, since every node of every ghost tree spanned the entire block. This is no longer the case, since block spanning ghost trees are now described by $\mathsf{WF}(\mathbf{t})$. Now we need to guarantee that the accessed range of offsets is within the domain of the ghost node at the access address. Thus we require clients to provide this evidence for the Hoare laws concerning block-spanning ghost trees, such as for retagging and freeing as shown in fig. 4.8.

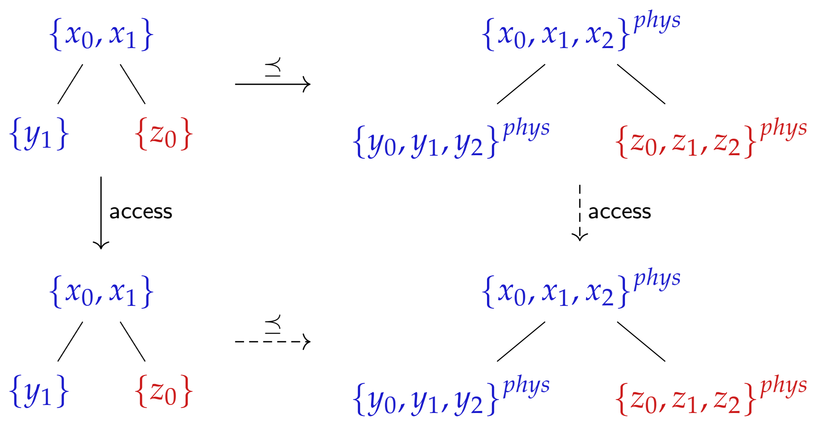

$\{x_0, x_1\}$
$\{y_1\}$
$\{z_0\}$
$\preceq$
$\{x_0, x_1, x_2\}^{phys}$
$\{y_0, y_1, y_2\}^{phys}$
$\{z_0, z_1, z_2\}^{phys}$
access
access
$\{x_0, x_1\}$
$\{y_1\}$
$\{z_0\}$
$\preceq$
$\{x_0, x_1, x_2\}^{phys}$
$\{y_0, y_1, y_2\}^{phys}$
$\{z_0, z_1, z_2\}^{phys}$

Figure 5.12: Case for the preservation proof under *lateral separation*. We illustrate the proof for an access to node y. This results in local accesses for nodes x and y, notated in blue, and a foreign access to z notated in red. The trees on the LHS represent *ghost trees* visible to users of the program logic, while the trees on the RHS represent *physical trees* owned by the underlying SI. We rely upon the fact that we may only access offset 1 at node y.

When proving preservation on the level of ghost nodes, we require that the set of accessed offsets is a subset of the block domain of the node only when the access is local. Consider fig. 5.12 from node x's perspective. In the ghost tree, node x contains block offsets $\{0, 1\}$. In order for the local access on node x to succeed, it is necessary to demonstrate that x's offsets contain the accessed offset 1, that $\{1\} \subseteq \{0, 1\}$. We are able to demonstrate this in general as a consequence of the well-formedness property. In this case for node x, we need to ensure that the access succeeds for the physical tree at node x for offset 1. Essentially, we reason that since we know that the access succeeds for node y at offset 1 in the ghost tree, and since 1 is included in x's offsets in the ghost tree, we may directly obtain evidence that the access succeeds for x at offset 1 in the physical tree.

But block domains for foreign-accessed nodes are not guaranteed to be supersets of the accessed range. Consider fig. 5.12 from node z's perspective. In the ghost tree, node z only has offset 0, and of course $1 \notin \{0\}$. Thus we need to demonstrate that the access succeeds and that the access preserves $\psi_{RHS}(z_1^{phys})$ for node z at offset 1 in the physical tree. Thankfully, this property follows from the definition of the Tree Borrows state machine.

Further research is required to uncover if and how this well-formedness property can be hidden from clients to the program logic. For now, we expose this to users, as shown in the points-to laws in fig. 4.9. In particular, in order to cleave apart a ghost tree one must show that each division is well-formed. In Rocq, we construct the witnesses for the partitions by filtering the offsets pointwise at each ghost node. These pointwise filters and explicit witnesses may be hidden from the user in some cases.

In the definition of $\ell \mapsto\!\!*_t \mathbf{t}$, we need to explicitly require $\mathsf{WF}(\mathbf{t})$ for the client-visible ghost tree as well as for the hidden, existentially quantified ghost tree in $\mathsf{WF}(\mathbf{t}')$, as shown in fig. 5.9. The weakening relation $\mathbf{t} \preceq \mathbf{t}'$ does not entail well-formedness in either direction.

5.4.2 Protector Agreement

$$\textsc{Protectors-Persistent}\quad bid \mapsto_p pt \vdash \square\, bid \mapsto_p pt$$

$$\textsc{Protectors-Agree}\quad bid \mapsto_p pt_1 * bid \mapsto_p pt_2 \vdash pt_1 \text{ and } pt_2 \text{ agree}$$

$$\textsc{Protector-Prefix}\quad \frac{pt_1 \subseteq pt_2}{bid \mapsto_p pt_2 \vdash bid \mapsto_p pt_1}$$

$$\textsc{Combine-Protectors}\quad bid \mapsto_p pt_1 * bid \mapsto_p pt_2 \vdash bid \mapsto_p pt_1 \cup pt_2$$

$$\textsc{Separate-Protectors}\quad \frac{pt_1 \text{ and } pt_2 \text{ agree}}{bid \mapsto_p pt_1 \cup pt_2 \vdash bid \mapsto_p pt_1 * bid \mapsto_p pt_2}$$

$$\textsc{Tree-Protectors-Agree}\quad \frac{\ell.\textsc{block\_id} = bid}{bid \mapsto_p pt * \ell \mapsto_t t \vdash pt \text{ and } (\textsc{prot} \lt\$\gt t) \text{ agree}}$$

$$\textsc{Grove-Protectors-Agree}\quad \frac{\ell.\textsc{block\_id} = bid}{bid \mapsto_p pt * \ell \mapsto\!\!*_t \mathbf{t} \vdash pt \text{ and } (\textsc{prot} \lt\$\gt \mathbf{t}) \text{ agree}}$$

Figure 5.13: Protector points-to laws.

Recall from the example shown in fig. 5.8 that we require some means of forcing protectors to agree across the blocks. Thus we introduce a new ghost map from the product of block identifiers and tree addresses to protector values. As shown in fig. 5.9, we essentially introduce an agreement map for protector values, as described in section 2.2.2. We define the points-tos for protectors $bid \mapsto_p pt$ in terms of *trees of protectors*, which we then store in the points-tos for both per-location and block-spanning trees. We use the protector values of the hidden, existentially quantified trees since these are the trees actually owned in the ghost state. We present the properties of the protector points-tos in fig. 5.13. Foremost, these points-tos are persistent, allowing us to duplicate and store them in the tree points-tos, as we have done with the block size points-tos.

The protector ghost state becomes necessary in order to prove the union laws Combine-Grove and Split-Grove for ghost trees. Recall the laws as shown in fig. 4.9. If we wish to combine $\mathbf{t}_1$ and $\mathbf{t}_2$ into $\mathbf{t}_1 \cup \mathbf{t}_2$, we must ensure that the protectors agree for their common nodes, which always includes the root. The attentive reader may inquire “why add more ghost state, why not simply add a constraint that the protectors agree for $\mathbf{t}_1$ and $\mathbf{t}_2$?” This indeed would constrain the protectors for $\mathbf{t}_1$ and $\mathbf{t}_2$, but not for the hidden, existentially quantified ghost trees $\mathbf{t}'_1$ and $\mathbf{t}'_2$.

Without a ghost state for protectors, there is no way to constrain the protectors for the existentially quantified trees, which in general are superstructures of their user-facing counterparts. $\mathbf{t}'_1$ and $\mathbf{t}'_2$ may share some structure outside of both $\mathbf{t}_1$ and $\mathbf{t}_2$, which is a blind spot of naively merely constraining the user-facing trees. By owning the protector ghost state for the concealed trees, we ensure that protectors match when we would like to combine ghost trees.

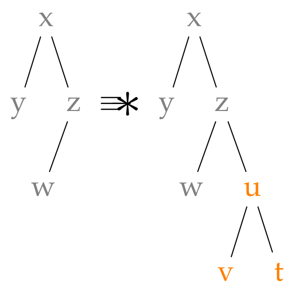


Figure 5.14: Performing an update for physical trees. We highlight the nodes from the old physical tree in gray, and we highlight the new nodes in orange. In order to provide the protector points-to for the updated tree points-to, we need to know that there is a path to node u in the new physical tree.

When we update the state interpretation with new trees, we must insert the new protector values using GhostMap-Auth-Insert-Persist-Big. Let the old physical tree be some t_1^{phys}, and the new physical tree be some t_2^{phys}. We require that the new tree is a superstructure of the old tree, that $\mathsf{dom}(t_1^{phys}) \subseteq \mathsf{dom}(t_2^{phys})$. Furthermore, the protector values of t_2^{phys} must agree with those of t_1^{phys}. In order to update the SI for protector values, we apply GhostMap-Auth-Insert-Persist-Big with new protector values $(\textsc{prot} \mathrel{<\$>} t_2^{phys}) \setminus (\textsc{prot} \mathrel{<\$>} t_1^{phys})$, which is trivially disjoint from the $(\textsc{prot} \mathrel{<\$>} t_1^{phys})$ already inhabiting the SI. In fig. 5.14, we describe the old, already present tree of protector values $(\textsc{prot} \mathrel{<\$>} t_1^{phys})$ with the gray nodes, and the new tree of protector values $(\textsc{prot} <\$> t_2^{phys}) \setminus (\textsc{prot} <\$> t_1^{phys})$ with the orange nodes.

Applying GhostMap-Auth-Insert-Persist-Big provides us with new fragment elements for $(\textsc{prot} <\$> t_2^{phys}) \setminus (\textsc{prot} <\$> t_1^{phys})$. Vexingly, we need the fragments for the entirety of $(\textsc{prot} <\$> t_2^{phys})$ in order to establish the new tree points-to fragment corresponding to t_2^{phys}. Recall from section 2.2.2 the "common trick" for agreement maps to ensure that every value that could have possibly been allocated into the ghost map is owned persistently. Note that we employ this technique for the protector ghost map, as shown in fig. 5.9. We may apply this technique to obtain the persistent fragments for the protector tree $(\textsc{prot} <\$> t_1^{phys})$, and combine this with our fragments from $(\textsc{prot} <\$> t_2^{phys}) \setminus (\textsc{prot} <\$> t_1^{phys})$ to recover $(\textsc{prot} <\$> t_2^{phys})$. Essentially, we need these owned fragments from the physical state in the SI to "bridge the gap" between the old ghost tree and the new ghost tree and physical tree.

5.4.3 Final Thoughts and Challenges

The inclusion of the well-formedness predicate (section 5.4.1) and protector agreement ghost state (section 5.4.2) introduce new case analysis and complexity into our development. In particular, the SI ghost update laws become much more challenging to prove, since for a block-spanning tree/grove we ultimately own and update per-location trees. This requires flexibility in our representations, where we may convert between trees of blocks and blocks of trees, and to and from relations between block spanning trees and relations between maps of per-location trees. Ultimately, this flexibility between representations pays off, since as a result we obtain laws such as WP-RetagN. In WP-RetagN, one may logically retag without owning all of the offsets in a block. In our Rocq implementation, we provide a version of WP-RetagN for per-location trees, derived from the more general law. Thus, for retagging, we demonstrate foundationally that it is sound to ignore unneeded offsets.

In order to reprove many results for lateral separation, especially for the update laws and safety and preservation, we break abstraction by considering what assumptions lemmas in the lower layers of our logic require of the primitive laws. In order to support lateral separation, it no longer suffices to perform exclusively pointwise reasoning about trees, and we need to constrain the tree structure and properties based on relative access locality.

Chapter 6

Conclusion

Summary of Contributions. We briefly recap our contributions:

- We produced accepted merge requests into Rocq-std++ and Iris list, vector, and finite map libraries.
- We built extensive Lilac Tree libraries.
- We implemented in Rocq a formalization of Tree Borrows built on Lilac Trees.
- We designed and constructed a mechanized Tree Borrows program logic in Iris featuring:
  - Pointwise permission weakening, enabling reconciliation of permissions between different branches of programs.
  - Subtree deletion, for curbing proof state explosion, especially for unused or disabled nodes.
  - Lateral tree separation, enabling concurrent separation and recombination of heap blocks.

6.1 Implementation

As is par for the course in Iris-based developments, we have proven *adequacy*[1] of our program logic. We have applied this adequacy result to our Rocq examples for `bar` (fig. 6.1) and `bongo` (fig. 4.13). In Rocq, we have more examples for our program logic emphasizing specific features (pointwise permission weakening, subtree deletion, lateral separation), and test cases for our implementation of the Tree Borrows state machine demonstrating that it exhibits all expected transitions.

[1]Our logical state via the state interpretation and points-tos and our notion of weakest preconditions faithfully reflect the physical execution of any program.

```rust
pub fn bar(opaque: impl FnOnce() -> bool + std::marker::Send) -> i32 {
    let mut root = Box::new([3, 3]);
    let w: &mut i32 = unsafe { &mut *std::ptr::addr_of_mut!(root[1]) };
    let (x0, x1) = root.split_at_mut(1);
    std::thread::scope(|s| {
        s.spawn(|| {
            let y: &mut i32 = &mut x0[0];
            if opaque() { *y = 42; } else { *y; }
        });
        s.spawn(|| { let z: &i32 = &x1[0]; *z; });
    });
    *w = 13; x0[0] = x0[0];
    let q = x0[0] + *w;
    // let r = &mut root[0..2];
    // drop(r);
    q
}
```

(a) Program bar in Rust. Two-phase borrows is constrained by the borrow checker to specific cases, such as implicit reborrows in function arguments. However, Tree Borrows does not impose these restrictions. Thus we exploit unsafe to bypass the borrow checker. This example succeeds under Miri when Tree Borrows is enabled, but trips UB under the default Stacked Borrows model.

$$
\begin{aligned}
\mathtt{bar}(\mathit{opaque}) = {} & \mathtt{let}\ \mathit{root} = \mathtt{AllocN}(2,3)\ \mathtt{in} \\
& \mathtt{let}\ w = \mathtt{Retag}(\mathit{root} +_{\mathrm{L}} 1, \mathtt{mut})\ \mathtt{in} \\
& \mathtt{let}\ x = \mathtt{Retag}(\mathit{root}, \mathtt{mut})\ \mathtt{in} \\
& \left(\begin{array}{l} \mathtt{let}\ y = \mathtt{Retag}(x, \mathtt{mut})\ \mathtt{in} \\ \mathtt{if}\ \mathit{opaque}(())\ \mathtt{then}\ y \leftarrow 42 \\ \ \mathtt{else}\ !y \end{array} \middle\| \begin{array}{l} \mathtt{let}\ z = \mathtt{Retag}(x +_{\mathrm{L}} 1, \mathtt{shared})\ \mathtt{in} \\ !z \\ \ \end{array} \right) \\
& w \leftarrow 13; x \leftarrow !x; \\
& \mathtt{let}\ q = !x + !w\ \mathtt{in} \\
& \mathtt{let}\ r = \mathtt{Retag}(\mathit{root}, \mathtt{mut})\ \mathtt{in} \\
& \mathtt{Free}(r); q
\end{aligned}
$$

(b) Program bar in λ^{TB}. We explicitly deallocate from the reference in r to demonstrate the flexibility of Tree Borrows. We have verified this example in Rocq and Iris with our program logic.

Figure 6.1: Large example for a Tree Borrows program logic. The function bar accepts a function opaque which (may) non-deterministically output true or false. The function allocates a new array on the heap, retags sibling mutable nodes, accesses different offsets in parallel, performs more operations on the original nodes, and then finally returns and frees memory. This example exhibits *two-phase borrows*, which enables the program to create sibling mutable references w and x, and creates a borrow tree both in the physical and logical state that is not virtually a list nor a stack.

Our Rocq implementation is $\approx$ 23500 lines[2] of code, which is distributed between greater than 11500 lines of specification and 11900 lines of Ltac proofs. The Lilac Tree and Tree Borrows program logic modules and libraries contribute the most to these numbers, but a significant contribution came from our utility modules that extend Rocq-std++ with more functions, lemmas, and tactics especially for finite maps, lists, and vectors. Our utility modules are not specific to our custom Lilac Tree implementation nor the program logic. Thus we submitted merge requests to Rocq-std++, Iris and Iris Contributions with lemmas that emerged from this work. In total across all three repositories, we have submitted 21 merge requests, and the Rocq-std++ and Iris maintainers accepted and upstreamed 14 of these. We list each of these accepted merge requests in chapter A.

Our implementation is available at `https://gitlab.inf.ethz.ch/ou-plf/tree-borrows-program-logic`, and the latest commit hash is ccca9f325f909dceed4fd88008862f64296dbfd9. Note that in our Rocq implementation, we do not call our core calculus λ^{TB}; we simply refer to it as "the core calculus".

6.2 Limitations

This section focuses upon limitations of *current features* and *design choices within our implementation*. For a summary of extensions that *go beyond the intended features of the implementation*, see section 6.3.

Exclusive Reference Tags Tree combining is not complete: we do not know that reference tags generated for separate trees of the same block are actually different. One possible solution entails adding ghost state for *exclusive tokens of a block identifier and tag*, and every retag generates a new exclusive token.

Zero-sized Retags To perform any retag, including retags with a range of 0, the operational semantics of our λ^{TB} requires that the block in question actually exists in the heap and has not been deallocated. Furthermore, the program logic requires ownership of the tree for the start offset of any retag, as shown in WP-RetagN. Systems programmers may want to create references to potentially deallocated blocks, to compute an array size for example. In Rust, it is sound to create a zero-sized reference to a deallocated block, but we do not model this in our operational semantics. Thus future work is required to better specify and handle these edge cases in our operational semantics and program logic.

[2]Only Rocq, no whitespace nor comments.

Relative Ghost Tree Offsets Offsets in block-spanning ghost trees/groves are global, rather than relative with respect to the offset in the location of the points-to. This is dissonant with value points-tos, which have offsets relative to the location's offset, as is common in many Iris-based program logics. The issue lies in constructing witnesses for lateral separation. As shown in fig. 4.7, ghost blocks at each node are indexed by natural numbers used for the offsets. This statically enforces that offsets may not be negative. In our global offset design, to construct a witness to partition a grove laterally for Split-Grove, we perform a filter operation for the left root domain and the right root domain. If we change the representation to use relative offsets, then we would need to perform a kmap on the keys of the right partition, subtracting them by the size of root domain of the left partition. However, in Rocq-std++ nearly all kmap lemmas require the operation on keys to be *injective in general*, not just injective for the domain of the given map. Subtraction on the natural numbers is not injective, thus in order for the proofs to be tractable, we would either need to change our definition of block-spanning trees to use integers for the block domains, or refine the Rocq-std++ libraries to support lemmas with weaker injectivity assumptions.

Exposed Memory Model In future work, we would prefer to entirely conceal the definition and construction of memory locations and references, which would scale up to more complex models with varying memory layouts and byte encodings.

Points-to Conversions for Single-location Trees Recall Tree-to-Grove from fig. 4.9. In this law, we may convert a per-location ghost tree into a grove spanning only a single offset. This grove must *exactly equal* the result of the conversion function $\mathcal{T}[\![t]\!]_{\ell.\text{OFFSET}}$. This law does not allow us to convert any grove for a single offset into a per-location tree. This is because groves may have *empty, spurious* nodes with *no block offsets*. While we may prune these vacuous nodes from a tree via Weaken-Tree, we may not add empty nodes into a tree, thus the bidirectional entailment does not hold. This is particularly cumbersome when reasoning about arbitrary trees, such as for fig. 4.13. When proving each thread for Bongo-Spec, we may not simply discard any potentially empty nodes, since the postcondition expects a tree with the same structure at the end. One possible solution involves modifying the weakening relation $\mathbf{t}_1 \preceq \mathbf{t}_2$ to allow nodes only on the left-hand side *if they have empty subblocks*. This solution may prove unfeasible, since this breaks intrinsic transitivity of $\mathbf{t}_1 \preceq \mathbf{t}_2$. Another solution may be to modify the well-formedness condition to prohibit such empty nodes. This should be sound, since in points-tos for block-spanning groves as shown in fig. 5.10, per-location trees constitute the underlying owned resource, which by definition cannot have these vacuous nodes. This would require modifying the proofs for Combine-Grove and Split-Grove, which construct witnesses of each partition of the tree via a filter on the offsets. Such a naive filter results in empty nodes, so if we modified well-formedness to forbid empty nodes, we would need to further refine the results of the filter to prune empty nodes. This limitation is at most annoying, and despite it we prove Tree-Accessor. The proof of Tree-Accessor required obtaining a "middle" per-location ghost tree, which was non-trivial given that this per-location tree may be a substructure of the overall original block-spanning grove.

6.3 Possible Extensions

This section offers *speculation* about directions for *new features* in future work.

Protectors In order to fully support protectors, we must change our operational semantics and program logic for λ^{TB}. The operational semantics needs to support *protected retags* and keep track of the call context in the vein of the Simuliris Tree Borrows Rocq implementation [31]. This entails that the operational semantics needs to strongly update certain protected nodes to unprotected at the end of some function calls. However, the current logical state would not be compatible with this, since protector values are currently placed in essentially an *agreement map*, which means that one cannot update a protector value since they are *persistent*. The program logic would still require some ghost state for protectors that enabled combining ghost trees, ensuring that nodes at common addresses have matching protector values. Perhaps some fractional permissions would be sufficient, but the fractions would vary *per node*, not per tree, making defining the new tree points-tos more difficult. Since an unprotected node remains unprotected, the ghost state could still enforce that unprotected values are persistent, but allow fractional permissions for protected nodes. If one laterally separates a tree with a protected node and then later wishes to execute the end of the call that strongly updates the node to Unprotected, then users would need to recombine the trees and the protected permissions would all need to sum to 1. Additional difficulties arise from the *protector end semantics*, which perform implicit accesses upon protector removal. This requires much more thought and consideration.

Interior Mutability In Rust, interior mutability is often achieved via use of `UnsafeCell<T>`, which allows multiple shared references to data that may be modified by exploiting `unsafe` Rust. Interior mutability is useful when one can only ensure the borrow checking rules at runtime. Rust developers employ interior mutability to implement data structures such as trees with parent pointers. Tree Borrows in Miri [31] already supports interior mutability, in particular with the special *reserved interior mutable* state.

To add this feature to λ^{TB}, one would need to add some interior mutable primitive to the syntax and semantics of λ^{TB}, and to update our Tree Borrows state machine with this special reserved permission. Interior mutability in Tree Borrows is under active development to refine its behavior and render it less permissive. Recently, Chen [3] developed more granular tracking to interior mutable data in Tree Borrows. We wait to adopt these features for interior mutability in λ^{TB} until the semantics become stable and edge cases are ironed out.

Vertical Tree Separation: Orphaning Our logic is capable of *laterally* separating trees across blocks, but it is not capable of *breaking apart the tree structure itself*. This would be a powerful reasoning principle, which would make specifications much more modular, as they perhaps would only need to be concerned with the *locally accessed* parts of a tree. For instance, a specification for a function accepting a reference, such as that for `bongo` (fig. 4.13) would not need a points-to for the entire tree, but just for a singleton tree representing the subtree starting from the given reference's tag. This would mean that a tree points-to would map a *reference* to a subtree, no longer a *location* to a tree. However, one must separate tree structure in way that guarantees *frame preserving updates*. For instance, consider separating a logical tree into a parent tree and a subtree (orphan) at some address. In general, performing Tree Borrows accesses on the orphaned subtree should result in *foreign* accesses to its parent tree. But naive separation of these trees would not incur that, and combining these trees back together may produce an invalid tree. Furthermore, performing accesses on some parts of the parent tree should result in local or foreign accesses to the orphaned tree. Thus vertical tree separation requires restrictions that respect the state machine. For instance, perhaps one may need restrict orphaning to cut only at addresses where any accesses to the orphaned tree only results in provably idempotent foreign accesses to the parent tree, and one forbids direct accesses to any part of the parent tree entirely.

Tree Resource Algebra In order to implement logical vertical tree separation/orphaning, we may require a *Resource Algebra (RA)* or *CMRA* for Lilac Trees. It is not clear how to fulfill the RA axioms for trees, especially how to instantiate the monoid composition $\cdot$ and core $|-|$ operations. For starters, we may need some notion of *addressed trees*, or trees paired with some address denoting their position from the root. The address in the pair would denote whether one tree is a subtree, parent tree, or cousin tree with respect to another. While one may intend this feature to support separating/orphaning a subtree from its parent tree, its implementation opens the door to reasoning about *combining cousin trees*. In Iris RAs, monoid composition $\cdot$ defines separation, and this operation needs to be total. The nature of combining cousin trees is an open research question. For example, one could define the RA for trees with optional data *Tree*$(K, A^?)$, and one could construct spurious parent nodes with $\perp$ data when combining cousin trees. One could also use these vacuous $\perp$ nodes when combining a parent tree with a orphan tree where there is a "gap" in the edge of the address domain of the parent tree and the address of the orphan-to-be-adopted tree. However, this is somewhat undesirable, since we create trees with empty, placeholder nodes. In terms of concrete implementation, it may be possible to encode a Lilac Tree RA in terms of the gmap RA. The protector ghost state, as shown in section 5.4.2, perhaps hints at the feasibility of such an encoding, where the underlying RA for protectors is a ghost map from block identifiers and tags to protector values, but the points-tos for protectors assign a block identifier to a tree of protectors. It may be possible to further refine this approach to achieve vertical separation for trees, and perhaps avoid entirely some of the aforementioned design pitfalls of directly defining a Lilac Tree RA.

Intermediate Node Deletion In Tree Borrows, the execution adds many nodes to the borrow trees as retags accrue over time. Many of these intermediate nodes are not directly accessed at some point for the rest of the execution, or become impossible to access, for instance if an invoked function performs retags to some passed in reference, and returns a final reference to the same location, as in fig. 4.13. The intermediate retags performed by such a function produce intermediate references that are not directly accessible to the rest of the program, but their grandchildren are, so we may not entirely delete their subtrees. Thus we would benefit from *logically deleting intermediate nodes* in the borrow trees. This would be especially helpful for verifying functions that recursively retag the same location. As with full subtree deletion section 5.3, this will likely require constraints to ensure that transitivity of the weakening relation $\mathbf{t}_1 \preceq \mathbf{t}_2$ and that safety and preservation (fig. 5.4) for trees still hold.

6.4 Related Work

Simuliris Tree Borrows Implementation Villani et al. [31] implements a core calculus for Tree Borrows in Rocq for Simuliris. This work handles protectors and interior mutability, which our λ^{TB} does not. Both Simuliris Tree Borrows and our λ^{TB} are limited to static retag sizes, whereas in real Rust and Miri, Tree Borrows retag sizes may be computed at runtime. We could have adopted its core calculus for this work, but we chose to implement Tree Borrows from scratch. Most importantly, we desired a more flexible tree data type than the one used in Simuliris, which employs a complicated encoding. We wanted the multi-child tree structure to be intrinsic to and evident from the tree's construction.

Stacked Borrows Program Logic Louwrink [20] features a program logic for Stacked Borrows [12] built in Iris. Theirs is the most similar work to ours, as it builds a core calculus and program logic sensitive to a candidate aliasing model for Rust. Obviously, the main difference between their work and ours is that their work targets Stacked Borrows, which misses some features from Tree Borrows, such as two-phase borrows and compatibility with block-based memory. Thus in Louwrink [20], there is only support for single-location allocations, ergo no lateral separation of blocks, which our work supports. However, Louwrink [20] supports *logical substacks*, which would correspond to both subtree deletion (which we support), and intermediate node removal (which we do not support). Louwrink [20] also support fractional stack points-tos, which one may use in instances where the access is idempotent. We do not currently support fractional tree points-tos, but we do not see any inherit obstacles to extending our framework to support this. Both Louwrink [20] and our work support pointwise permission weakening, which they include in their *substack relation*. While our work does not fully support protectors, Louwrink [20] does not address protectors in any layer. Our support for protectors is indeed incomplete in our work, but we hope it becomes a solid foundation for future work, where especially the RHS-only condition in our weakening relation will likely endure the refactors to fully support protectors.

RustBelt and λ_{Rust} RustBelt [13] via its λ_{Rust} formalization includes a semantic typing relation built upon a *lifetime logic*, which is meant to capture Rust borrowing behavior, especially for `unsafe` contexts. However, critically, RustBelt *does not* consider any aliasing model, rendering its results potentially incompatible with some compiler optimizations. In moving towards a more complete Rust formalization and program logic, one may want to develop a new version of RustBelt with logical relation for a Tree Borrows-based language, that is if the Rust community adopts Tree Borrows as the definitive aliasing model for Rust. Tree Borrows is likely a better choice than Stacked Borrows, since Tree Borrows considers more features such as dereferencing from pointer arithmetic outside the retag range and two-phase borrows, and Stacked Borrows is incompatible with some important compiler optimizations, such as read-read reorderings. Thus a new λ_{Rust} formalization and lifetime logic may need to capture the behavior of Tree Borrows. The SOS of λ_{Rust} would need to adopt Tree Borrows, and the lifetime logic and the proof of the *the fundamental theorem of logical relations* for λ_{Rust} would likely need to be updated and refactored as well.

VeriFast Tree Borrows Proposal VeriFast [33] is an automated verification framework for stateful and multithreaded programs based on separation logic. Jacobs [10] proposes a set of rules to verify programs against Tree

Borrows in VeriFast. Unlike our work, their proposal has not yet been proven sound nor adequate with respect to Tree Borrows. A key difference between our work and theirs is that they do not have an explicit logical tree points-to. Rather, their logical connectives appear to encode Tree Borrows with the separate points-tos for each node, tokens determining whether nor not there is permission to end a reference, and connectives that signify parent-child relationships between nodes in a tree. Furthermore, Jacobs [10] does not feature explicit Tree Borrows permissions. It is unclear how their logical connectives and laws exactly relate to the Tree Borrows state machine. Jacobs [10] claims to support protectors and interior mutability, which was out of scope for our work. In general, their proposal seems to be sound with respect to Stacked Borrows as well. This likely entails that their proposal does not support two-phase borrows via the Reserved permission, nor pointer arithmetic outside of the explicit retag bounds, which we support. In future work, it may be possible to build a soundness proof for the laws in Jacobs [10] based on our framework.

Bibliography

[1] Umut Acar and Daniel Sleator. 2025. Binary Search Trees. In *15-210 Parallel and Sequential Data Structures and Algorithms - Notes*. Carnegie Mellon University. https://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/index.html Lecture Notes for 15-210.

[2] Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. *Proc. ACM Program. Lang.* 9, POPL, Article 79 (Jan. 2025), 27 pages. doi:10.1145/3704915

[3] Xinglu Chen. 2025. Adding Finer-Grained Tracking of Interior Mutability to Tree Borrows. Bachelor's thesis, ETH Zurich, Zurich, Switzerland.

[4] Athan Clark. 2025. rose-trees: Various trie implementations in Haskell. https://hackage.haskell.org/package/rose-trees

[5] MiniRust Contributors. [n. d.]. MiniRust. https://github.com/minirust/minirust

[6] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. doi:10.1145/3371102

[7] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 192 (June 2024), 25 pages. doi:10.1145/3656422

[8] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL, Article 28 (Jan. 2022), 31 pages. doi:10.1145/3498689

[9] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 247 (Oct. 2023), 29 pages. doi:10.1145/3622823

[10] Bart Jacobs. 2024. vf-rust-aliasing. https://github.com/btj/vf-rust-aliasing. f4daea709e32734f43b9735b5fa30d330332c570.

[11] Ralf Jung. 2024. Miri: Practical Undefined Behavior Detection for Rust (Keynote). In *Proceedings of the 19th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems* (Vienna, Austria) *(ICOOOLPS 2024)*. Association for Computing Machinery, New York, NY, USA, 1. doi:10.1145/3679005.3695733

[12] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages. doi:10.1145/3371109

[13] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154

[14] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.

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

[16] 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.

[17] Robbert Krebbers. 2023. Efficient, Extensional, and Generic Finite Maps in Coq-std++. In *Coq Workshop*.

[18] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.* 52, 1 (Jan. 2017), 205–217. doi:10.1145/3093333.3009855

[19] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

[20] Daniël Louwrink. 2021. *A Separation Logic for Stacked Borrows*. Master's Thesis. Universiteit van Amsterdam. https://eprints.illc.uva.nl/id/eprint/1790/

[21] William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 6 (Jan. 2024), 27 pages. doi:10.1145/3632848

[22] 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 *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. doi:10.1145/3453483.3454036

[23] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (Jan. 2023), 31 pages. doi:10.1145/3571220

[24] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025. Formal Semantics and Program Logics for a Fragment of OCaml. *Proc. ACM Program. Lang.* 9, ICFP, Article 240 (Aug. 2025), 32 pages. doi:10.1145/3747509

[25] The std++ team. [n. d.]. Rocq-std++. https://gitlab.mpi-sws.org/iris/stdpp. 773a75ac897523c6c6fbfd6cbbb6a2611452c396.

[26] The Iris Team. [n. d.]. Iris. https://gitlab.mpi-sws.org/iris/iris. c5014d246b2cc5d1bf79d3ba362501dd7b447f74.

[27] The Iris Team. 2024. The Iris 4.3 Reference. https://iris-project.org/.

[28] The Rust Project Developers. 2025. The Rust Programming Language. https://doc.rust-lang.org/book/title-page.html. Accessed: 2025-09-16.

[29] The Rust Project Developers. 2025. Rust Unsafe Code Guidelines Reference. https://rust-lang.github.io/unsafe-code-guidelines/introduction.html. Accessed: 2025-09-11.

[30] Andrea Vedaldi. 2025. Binary trees. In *B16 Algorithms and Data Structures 1 - Notes*. University of Oxford. https://www.robots.ox.ac.uk/~vedaldi/assets/teach/2024/b16/notes/4-binary-trees.html Lecture Notes for B16.

[31] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. doi:10.1145/3735592

[32] Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 76–90. doi:10.1145/3437992.3439930

[33] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Log. Methods Comput. Sci.* 11, 3 (2015). doi:10.2168/LMCS-11(3:19)2015

Appendix A

Upstreamed Contributions

Here we list the upstreamed merge requests that resulted from this work. We order them by the merge date for each repository.

Rocq-std++

- MR #599: Vector Forall lemmas.
- MR #611: fmap imap compose.
- MR #609: kmap lemmas for map_to_list and list_to_map.
- MR #618: list_to_set auxiliary lemmas.
- MR #612: map_seq and map_seqZ auxiliary lemmas.
- MR #614: Lemma map_to_list_update.
- MR #603: Lookup lemmas for union list over finite maps.
- MR #620: Forall exists Forall2 lemmas.
- MR #596: Zip list lemmas.
- MR #610: Finite map alter lemmas.

Iris

- MR #1113: Fix HeapLang documentation typo.
- MR #1116: big_opL zip seq lemmas.
- MR #1131: big_op for kmap, map_seq, and map_seqZ.

Iris Contributions

- MR #5: Finite map transposition.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

[✓] I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

A Program Logic for Tree Borrows

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Peterson	Rudy

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 19.09.2025	[illegible]

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).