Precise Reasoning about Container-Internal Pointers with Logical Pinning

Yawen Guan
EPFL
Lausanne, Switzerland
yawen.guan@epfl.ch

Clément Pit-Claudel
EPFL
Lausanne, Switzerland
clement.pit-claudel@epfl.ch

Abstract

Most separation logics hide container-internal pointers for modularity. This makes it difficult to specify container APIs that temporarily expose those pointers to the outside, and to verify programs that use these APIs.

We present *logical pinning*, a lightweight borrowing model for sequential programs that allows users to selectively track container-internal pointers at the logical level. Our model generalizes the magic-wand operator for representing partial data structures, making it easy to write and prove precise specifications, including pointer-stability properties. Because it only changes the way representation predicates and specifications are written, our approach is compatible with most separation logic variants.

We demonstrate the practicality of logical pinning by verifying small but representative pointer-manipulating programs, and deriving more precise versions of common container specifications. In doing so, we show that our approach subsumes some well-known proof patterns, simplifies some complex proofs, and enables reasoning about program patterns not supported by traditional specifications. All of our results are mechanized in the Rocq proof assistant, using the CFML library.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**.

Keywords: Representation predicates, Containers, Sharing

ACM Reference Format:
Yawen Guan and Clément Pit-Claudel. 2026. Precise Reasoning about Container-Internal Pointers with Logical Pinning. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779031.3779096

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779096

1 Introduction

Separation logic [10, 24–26] is an extension of Hoare logic for modular verification of pointer-manipulating programs. A central challenge in reasoning about such programs is aliasing, since mutating a shared memory cell may affect the value of syntactically unrelated expressions.

Separation logic helps avoid this challenge by enabling assertions over disjoint heap fragments using the separating conjunction operator ⋆, such that mutating a single memory cell in the heap affects only one ⋆-conjunct. The primitive heap proposition in the logic is $p \mapsto v$, asserting exclusive *ownership* (permission to access and mutate) of the cell at location p storing data v. A ⋆-conjunction like $p \mapsto v \star q \mapsto u$ ensures that p and q are not aliases: the ⋆ captures the absence of sharing, enforcing non-aliasing by construction.

In separation logic, mutable containers such as linked lists and trees are described using predicates, called *representation predicates*, which relate the container's logical model to a concrete heap representation. For modularity, most of them hide the container's internal pointers and assert full ownership of the container and its content.

For example, consider a linked list whose nodes are represented in C as:

```
struct lnode { elem x; struct lnode* next; };
```

Assume that the C type `elem` corresponds to the logical type A, and that the representation predicate $R_A\ x\ p$ asserts that an element x of type A is stored at p. The typical representation predicate of such lists, List, can be defined as:

$$\mathsf{List}\ R_A\ []\ p \triangleq \lfloor p = \mathtt{null} \rfloor$$
$$\mathsf{List}\ R_A\ (x :: l)\ p \triangleq R_A\ x\ p \star \exists q.\, p + 1 \mapsto q \star \mathsf{List}\ R_A\ l\ q$$

This asserts that the pointer p is `null` when the list is empty, and otherwise the head x is stored at location p and a pointer q pointing to the tail l is stored at $p + 1$. The recursive use of ⋆ enforces that all list nodes occupy disjoint heap fragments. Such representations are convenient and effective when the non-aliasing assumption holds, namely, that each element is accessible only through a unique path rooted at the container's entry point.

Unfortunately, the non-aliasing assumption often breaks down in practice: container interfaces and implementations commonly expose internal pointers to elements (stored items) or substructures to the outside. Graphs are inherently built

on sharing, whereas other containers expose internal pointers only temporarily, either to client code or during internal operations such as tree rebalancing. In this work, we focus on containers with transient internal-pointer exposure, both flat (e.g., arrays and records) and recursive (e.g., linked lists and trees), and develop techniques for reasoning about them in sequential programs written in pointer-manipulating languages such as C.

To illustrate the challenges concretely, consider the following logger example, inspired by the APIs of the C++ logging library `spdlog` [2] and rewritten in C for simplicity. A logger is an object that maintains a configuration, such as log levels, and directs messages to designated outputs. Applications often manage a dictionary of named loggers; in this example, `registry` is a pointer to such a dictionary, whose loggers can be updated individually or collectively.

```
1 struct logger* lgr = get(registry, k);
2 // spdlog::set_level
3 set_level_all(registry, INFO);
4 // spdlog::logger::set_level
5 set_level(lgr, DEBUG);
```

Line 1 retrieves a pointer to the logger associated with key `k` and stores it in variable `lgr`; line 3 sets the log level of *all* loggers in `registry` to `INFO`; and line 5 overrides the level of the logger pointed to by `lgr` to `DEBUG`. This program is safe because the implementation of `set_level_all` ensures pointer stability: it updates loggers in place, so the pointer `lgr` remains valid for use on line 5.

Conceptually, `set_level_all` is a map operation that updates all loggers in the dictionary. Let Dict $d\ p$ denote the representation predicate for a dictionary d stored at p, and $g[\mathsf{lvl} := v]$ denote the logger g with its level substituted by v. A typical specification of `set_level_all` is:

$$\forall\ d\ p\ v.\ \{\ \mathsf{Dict}\ d\ p\ \}\ \texttt{set\_level\_all}(p, v)\ \{\ \lambda\ \_.\ \mathsf{Dict}\ (\mathsf{map}\ (\lambda\ g.\ g[\mathsf{lvl} := v])\ d)\ p\ \}$$

Unfortunately, this specification is insufficient to verify the program: it does not capture pointer stability, and hence permits implementations that relocate (move) loggers and invalidate exposed internal pointers such as `lgr`.

Existing approaches fall short on verifying this pattern for various reasons (section 3). In this paper, we introduce a *logical-pinning* model for reasoning about exposed internal pointers, together with a *proof discipline* in which the user selectively exposes such pointers in the logical representations and precisely tracks pointer validity in API specifications.

Section 2 outlines the key ideas of our approach. Section 3 discusses the limitations of existing state-of-the-art approaches. Section 4 presents the logical-pinning model and proof discipline. Sections 5 and 6 apply the model to a series of containers and discuss the results, showing that it subsumes existing ad hoc proof patterns, allows complex proofs to be simplified, and enables verification of new program patterns. We conclude with related work in section 7.

Contributions.

- We introduce *logical pinning*, a model and proof discipline for precise reasoning about exposed internal pointers in containers.
- We showcase applications of the model in a series of case studies, covering containers ranging from single cells to arrays, records, linked lists, and trees.
- We show that logical-pinning enables maximally precise specifications for realistic APIs by specifying a realistic, low-level API for linked lists.
- We show that logical pinning is easy to implement in practice on top of existing separation logics by formalizing all of our examples using the CFML library.

2 Overview and Key Ideas

Many programming idioms require containers to temporarily expose their internal pointers to the outside:

- Pointer caching, to avoid repeated lookups in performance-critical code;
- Direct manipulation, such as swapping two values stored in a list after retrieving pointers to them;
- Transient cross-container sharing, such as binary-tree rotation (fig. 1) where a subtree t_{rl} is temporarily shared between two trees.

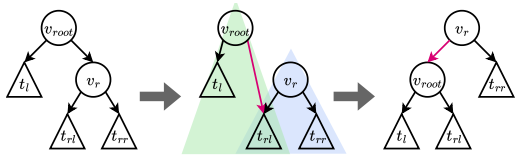

Figure 1. Left rotation of a binary tree.

Exposing internal pointers raises two challenges: (1) reasoning about their validity under container operations, and (2) supporting *multiple borrows*, where ownership of several subparts referenced by exposed pointers may be temporarily detached from the container for local reasoning. Existing approaches, such as magic wands, fall short on resolving these challenges and are discussed further in section 3.

2.1 Key Idea: Capture Pointer Stability with a Precise Classification of Writes

Pointer stability is a property of a container operation that ensures pointers to its subparts remain valid, i.e., the subparts are neither deallocated nor relocated. This property allows clients to safely cache and reuse pointers across operations.

The traditional read-write model, which broadly classifies memory operations as reads and writes, is too coarse to reason about pointer stability because it fails to distinguish between fundamentally different types of writes:

- ***In-place writes***: Writes that perform in-place mutation, such as the `set_level_all` operation. They change the content without invalidating existing pointers.
- ***Full writes***: Writes that may deallocate or relocate objects, such as resizing, rebalancing, or restructuring containers. They can invalidate previously exposed pointers.

Many container APIs (e.g., in C++ STL) explicitly document whether an operation invalidates internal pointers [1]. Correspondingly, we would like to capture in specifications which kind of writes a function may perform. Consider a container c consisting of four elements $e_1 \dots e_4$, where only two internal pointers are exposed: p_1 to e_1 and p_3 to e_3 (left of fig. 2; the right is discussed in section 4.3.3). For readability, we write $p \xrightarrow{R} x$ for application $R\ x\ p$ in figures. Conceptually, any function f that manipulates c while promising to preserve these exposed pointers must treat e_1 and e_3 as *pinned* at their location, meaning f may only perform reads and in-place writes to them. In contrast, e_2 and e_4 are considered *floating*; f may perform full writes to them.

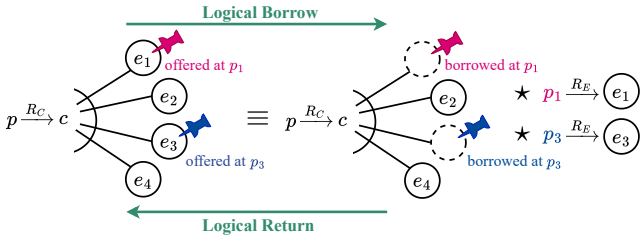

Figure 2. Two views of a container c with elements $e_1 \dots e_4$ and exposed internal pointers p_1 and p_3. Left: c owns all elements. Right: e_1 and e_3 are borrowed from c.

2.2 Key Idea: Encode Representation Predicates as Tagged Multi-Hole Contexts

In practice, programs often maintain multiple exposed pointers to container subparts (elements and substructures) rather than just one. A simple example is the *element-swapping* program which retrieves pointers to two list elements and swaps their contents. When ownership of these subparts is detached for local reasoning (i.e., these subparts are *borrowed*), the container is left with multiple holes.

Consider a key-value tree t pointed to by p. The program below looks up potentially overlapping subtrees t_1 and t_2 rooted at keys k_1 and k_2 before passing them on to `compare`:

```
1 struct tree* q1 = lookup(p, k1);
2 struct tree* q2 = lookup(p, k2);
3 compare(q1, q2);
```

A typical specification for `lookup`(p, k_1) requires the full ownership of t, and splits that ownership between t_1 (rooted at q_1) and a partial tree rooted at p. This partial tree is often represented using a custom predicate or a magic wand.

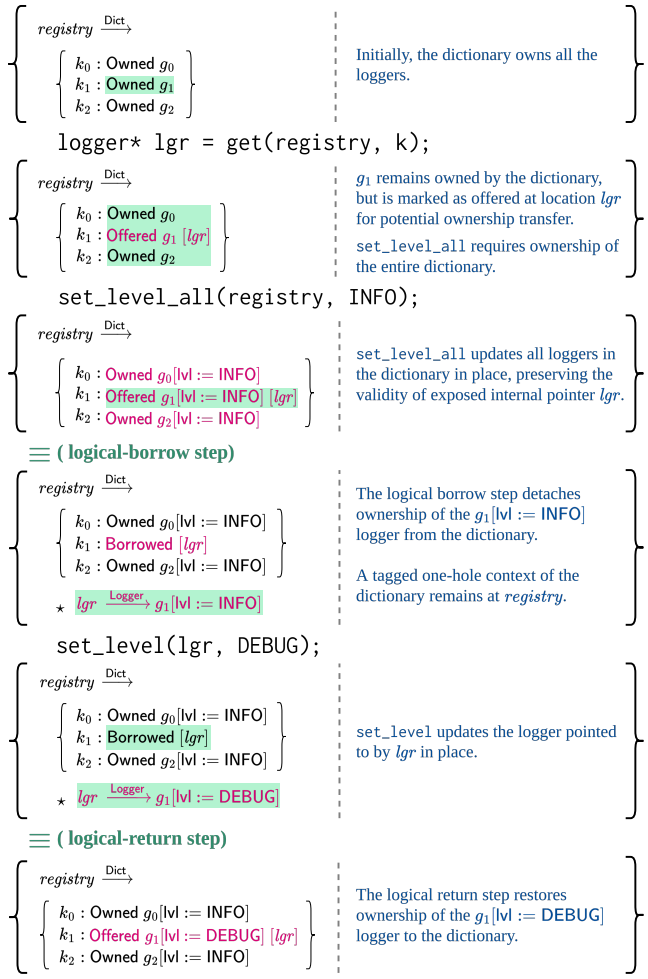

Figure 3. An informal proof sketch of the functional correctness of the logger example. For simplicity, we assume the dictionary initially contains three key-logger pairs (k_0, g_0), (k_1, g_1) and (k_2, g_2), where $k = k_1$. Each step is annotated with the corresponding abstract heap state, denoted $\{\ state\ \}$, together with a brief informal explanation (shown in blue). $[lgr]$ denotes an alias list with a single element lgr. Color markers: to change, just changed.

This specification performs two logical steps at once: it records the subtree relation between q_1 and p, and transfers ownership of t_1 to q_1. This ownership transfer is incompatible with the second lookup, which requires p to have full ownership of t.

Our approach distinguishes these two steps: in it, the calls to `lookup` pin t_1 to q_1 and t_2 to q_2 without detaching them, and without changing the predicate used to represent t. Later, when separate ownership is needed to call `compare`, we add a *logical borrowing* step to detach t_1 and t_2.

To do so, we adopt a uniform representation for whole and partial containers using *tagged multi-hole contexts*, in

which each borrowable subpart is augmented with a fine-grained *borrowing state* that indicates whether the subpart is pinned or floating, and whether the container still has its ownership. This allows precise specifications that support flexible pointer tracking and ownership transfers for individual subparts without breaking the container abstraction.

This subtree-compare example is verified using our approach; the key to handling potential overlapping subtrees is the use of non-separating conjunction (section 3.1) in the precondition of `compare`. See section A for details.

2.3 The Logger Example Revisited

Figure 3 presents an informal correctness proof of the logger example under our approach. The predicate Logger $g\ p$ asserts that the logger g is stored at p. We write specifications with implicit universal quantification over free variables (e.g., g, p, v below). The specification of `set_level` is:

$$\{\,\mathsf{Logger}\ g\ p\,\}\ \texttt{set\_level}(p, v)\ \{\,\lambda\ \_.\ \mathsf{Logger}\ g[\mathsf{lvl} := v]\ p\,\}$$

We tag each logger in the dictionary with one of the four borrowing states: Owned, Offered, Borrowed or Missing. We refine the specification of `get` to mark the returned pointer *lgr* by offering the owned logger g_1 at *lgr*, essentially pinning it without detaching its ownership from the dictionary. We update the specification of `set_level_all` to preserve all pinned locations in the dictionary, ensuring the validity of exposed internal pointers.

3 Background

In this section, we first briefly introduce separation logic, then review and discuss the limitation of the main state-of-the-art approaches for reasoning about access to and mutation of container subparts.

3.1 Separation Logic

Separation logic is a refinement of Hoare logic that enables local reasoning: one can prove a Hoare triple by describing only the relevant heap fragment and apply the triple in a larger context. We use a standard shallow embedding of a linear separation logic in a higher-order logic, which models heaps (heap fragments) as partial functions from locations to values, with type loc → val (abbreviated as heap).

We assume the existence of a distinguished `null` pointer and allow pointer arithmetic on locations, so an expression like $p + n$ denotes the location n steps offset from the base location p. We denote the empty heap as $\emptyset$. Two heaps h_1 and h_2 are disjoint (written $h_1 \perp h_2$) when their domains do not overlap, and can be combined by union, denoted as $h_1 \uplus h_2$.

A heap predicate H is a function of type heap → prop (abbreviated as hProp), used to describe properties of heaps. Figure 4 shows the basic heap predicate combinators. $\lfloor P \rfloor$ asserts a pure (non-spatial) proposition P. $p \mapsto v$ denotes a singleton heap mapping location p to value v. The separating conjunction $H_1 \star H_2$ holds when the heap can be divided into

$$\begin{aligned}
\lfloor P \rfloor &\triangleq \lambda\, h.\ h = \emptyset \ \wedge\ P \\
p \mapsto v &\triangleq \lambda\, h.\ h = (p \to v) \ \wedge\ p \neq \mathtt{null} \\
H_1 \star H_2 &\triangleq \lambda\, h.\ h_1 \perp h_2 \ \wedge\ h_1 \uplus h_2 \ \wedge\ H_1\ h_1 \ \wedge\ H_2\ h_2 \\
\exists\, x.\, H &\triangleq \lambda\, h.\ \exists\, x.\ H\ h \qquad \forall\!\!\forall\, x.\, H \triangleq \lambda\, h.\ \forall\, x.\ H\ h \\
H_1 \mathrel{-\!\!\star} H_2 &\triangleq \exists\, H.\ H \star \lfloor (H_1 \star H) \vdash H_2 \rfloor \\
&\qquad \text{where } H_1 \vdash H_2 \triangleq \forall\, h.\ H_1\ h \to H_2\ h \\
H_1 \mathbin{\&\!\&} H_2 &\triangleq \lambda\, h.\ H_1\ h \wedge H_2\ h \qquad H_1 \mathbin{\Vbar} H_2 \triangleq \lambda\, h.\ H_1\ h \vee H_2\ h
\end{aligned}$$

Figure 4. Basic heap predicate combinators.

two disjoint parts satisfying H_1 and H_2 respectively. $\exists\, x.\, H$ and $\forall\!\!\forall\, x.\, H$ are existential and universal quantification over values in the heap ($\star$ has higher precedence than $\exists$ and $\forall\!\!\forall$). The magic wand (separating implication), $H_1 \mathrel{-\!\!\star} H_2$, denotes a heap that, when extended with any heap satisfying H_1, yields a heap satisfying H_2. The non-separating conjunction $H_1 \mathbin{\&\!\&} H_2$ asserts that the heap satisfies H_1 and H_2, while the disjunction $H_1 \mathbin{\Vbar} H_2$ asserts it satisfies at least one of them.

We specify programs using Hoare triples of the form $\{\,H\,\}\ e\ \{\,\lambda\, v.\ Q\ v\,\}$ where e is an expression, H is the precondition describing the heap before execution, v is the return value of e, and $Q\ v$ is the postcondition describing the heap after the execution of e. This triple asserts total correctness: if the initial heap satisfies H, then e will terminate and produce an output value v and a heap fragment satisfying $Q\ v$.

A representation predicate R_A relates the logical model of an object with a location (its entry point) and a heap fragment (the memory it occupies). Formally, R_A has type $A \to \mathsf{loc} \to \mathsf{hProp}$, abbreviated as repr A.

3.2 Magic Wands

In separation logic, detaching ownership (borrowing) is traditionally modeled using a magic wand $\mathrel{-\!\!\star}$. This operator is well-suited for a single immutable borrow, where the borrowed subpart can later be returned unchanged.

In the logger example, a possible but unhelpful specification of `get` tracks the returned pointer using a magic wand:

$$\{\,\lfloor d[k] = g \rfloor \star \mathsf{Dict}\ d\ p\,\}\ \texttt{get}(p, k)\ \{\,\lambda\, r.\ \mathsf{Logger}\ g\ r\ \star\ (\mathsf{Logger}\ g\ r \mathrel{-\!\!\star} \mathsf{Dict}\ d\ p)\,\}$$

$d[k]$ denotes the logger associated with key k stored in dictionary d. This specification is limited: after performing `get`, operations that require the dictionary representation Dict $d\ p$, such as `get` and `set_level_all`, cannot be performed until the current wand is canceled. However, wand cancelation logically forgets the exposed pointer and thus invalidates it. Moreover, if the borrowed logger is mutated, the wand cannot be canceled at all, leaving the dictionary inaccessible.

$$\textit{(wand cancellation)} \quad H_1 \star (H_1 \mathrel{-\!\!\star} H_2) \vdash H_2$$

Therefore, when verifying the logger example with this specification, *lgr* is invalid after calling `set_level_all`, causing the verification to get stuck at `set_level(lgr, DEBUG)`.

Magic wand as frame. The *magic-wand-as-frame* [8, 29] technique extends the traditional use of magic wand to support a single mutable borrow. Instead of fixing the borrowed value, it universally quantifies over all possible updates, allowing the value to be modified and later reintegrated.

Under this approach, `get` can be specified as:

$$\{ \lfloor d[k] = g \rfloor \star \mathsf{Dict}\ d\ p \}\ \texttt{get}(p,\ k)$$
$$\{ \lambda\, r.\ \mathsf{Logger}\ g\ r \star (\forall\, g'.\ \mathsf{Logger}\ g'\ r \mathrel{-\!\!\star} \mathsf{Dict}\ d[k := g']\ p) \}$$

$d[k := g']$ denotes the dictionary d with the logger associated with key k substituted by g'.

This specification allows the borrowed logger g to be updated in place and returned to the dictionary afterward. However, the magic-wand-as-frame technique still does not support multiple borrows: a second call to `get` again requires canceling the wand and thereby revoking the first borrow.

3.3 Trees with Holes and Cut Subtrees

Charguéraud [9] introduces a technique to take multiple mutable borrows of elements or subtrees within a tree. Consider a binary tree whose nodes are represented in C as:

```
struct tnode { elem x; struct tnode *lt, *rt; };
```

In addition to the standard constructors for leaves and nodes, datatype $\mathsf{tree}_{\mathsf{hc}}$ includes two additional constructors:

$$\begin{aligned}\mathsf{tree}_{\mathsf{hc}}\ A \ \triangleq\ & \mathsf{Leaf} \mid \mathsf{Node}\ (x : A)\ (t_l, t_r : \mathsf{tree}_{\mathsf{hc}}\ A) \\ & \mid \mathsf{Hole}\ (q : \mathsf{loc})\ (t_l, t_r : \mathsf{tree}_{\mathsf{hc}}\ A) \mid \mathsf{Cut}\ (q : \mathsf{loc})\end{aligned}$$

Node $x\ t_l\ t_r$ denotes a node with element x, left and right subtrees t_l and t_r; Hole $q\ t_l\ t_r$ denotes a node whose element stored at q is absent, while the subtrees remain owned; and Cut denotes an absent subtree at q.

The type $\mathsf{tree}_{\mathsf{hc}}$ models trees with possibly absent elements (holes) and subtrees (cuts), and thus provides a uniform representation for whole and partial trees. Each hole and cut is annotated with its location to enable reattachment. The tree representation is the same as for ordinary trees, except that if a tree stored at p is Hole $q\ t_l\ t_r$ or Cut q, then $p = q$ and the ownership of the absent element or subtree is not held.

This representation models ownership transfer in a single step: an element or subtree is either present or replaced by a hole or a cut. However, this granularity is too coarse for patterns such as the subtree–compare example in section 2.2. There, specifying $\texttt{lookup}(p,\ k_1)$ with this representation requires converting t_1 into a cut to track its pointer q_1, thereby detaching its ownership from t. This detachment is undesirable, since it either (1) blocks a second `lookup`, or (2) forces the restoration of t_1 into t, which hides the location of t_1 behind the representation of t and thus logically forgets and invalidates q_1 before it is used in `compare`.

3.4 Borrows in Rust

The pattern of exposing and manipulating internal pointers of containers is common in low-level programming. The same pattern naturally arises in Rust, where the type system enforces memory safety through controlled aliasing.

We return in section 7 to Rust's borrowing model, including the standard borrows, splitting borrows and deferred borrows. While these mechanisms provide strong safety guarantees, they remain insufficient to express certain patterns of internal pointer exposure: current Rust cannot easily express the above examples (logger updates, element swaps, or borrowing multiple elements and subtrees in a tree). The Rustonomicon calls our element-swapping example "pretty clearly hopeless" for "general container types like a tree" [28].

4 The Logical-Pinning Model for Reasoning About Internal Pointers

In this section, we present *logical pinning*, a lightweight borrowing model for reasoning about sequential programs that expose and manipulate pointers to container subparts (i.e., elements and substructures). At its core, the model defines *fine-grained borrowing states* along with transitions that describe how these states evolve in response to program operations or logical reasoning steps.

Building on this model, we introduce a proof discipline in which users (1) identify borrowable subparts and redefine the container's representation predicate to track their borrowing states; (2) enrich API specifications to reflect the validity of exposed internal pointers; and (3) prove correctness of API and client programs with explicit logical borrow and return steps around subpart access. Our technique is formulated entirely within the basic form of separation logic defined in section 3.1, and does not require language-level extensions.

4.1 Informal Overview

Each borrowable subpart has two orthogonal properties: (1) whether this subpart is *available* to the container owner, i.e., the container owner has ownership of this subpart and therefore can access its logical value v; (2) whether it is *pinned* at an exposed memory location q. The combination of these properties yields four borrowing states for each borrowable element or substructure of a container:

- **Owned v**: The subpart is available and unpinned. The container owner may freely relocate it without affecting its logical state.
- **Offered v $(q :: qs)$**: The subpart is available but pinned to a specific exposed location q. The list qs (possibly empty) represents additional exposed aliasing pointers.
  In this state, the container owner retains ownership of this subpart, but exposes its location for others to borrow it in the future. To preserve the same exposed location, the owner may perform only in-place updates, not relocation.
- **Borrowed $(q :: qs)$**: The subpart is pinned to the exposed location q (with additional aliases qs) and currently borrowed by another party. The subpart is unavailable to the container owner until it is returned at the same location.

• **Missing**: The subpart is unavailable and not tracked.

The Offered and Borrowed states are particularly important for modeling ownership transfer. One can decompose a container c that holds offered elements as follows:

$$\begin{aligned} & \mathrm{c}\ \textit{with element}\ \mathrm{e}\ \textit{offered at}\ \mathrm{q}\ \textit{is stored in memory} \\ \equiv\ & \mathrm{c}\ \textit{with an element borrowed at}\ \mathrm{q}\ \textit{is stored in memory} \\ & \quad \star\ \mathrm{e}\ \textit{is stored at}\ \mathrm{q} \end{aligned}$$

The forward direction corresponds to a *logical borrow*, and the backward direction to a *logical return*.

We introduce the borrowable datatype to lift a base logical type A to its borrow-aware form borrowable A (written ${}^{\flat}A$) that models the four borrowing states described above. We also define a predicate transformer liftToB that lifts the corresponding base representation predicate R_A to liftToB R_A (written ${}^{\flat}R_A$) that describes borrowable objects of type ${}^{\flat}A$. Full definitions are provided later in section 4.3.1.

	type	representation predicate
base object	A	R_A : repr A
borrowable object	${}^{\flat}A$	${}^{\flat}R_A$: repr ${}^{\flat}A$

4.2 Step-by-Step Guide

We illustrate the use of our model and proof discipline step by step using the motivating logger example introduced in section 1. For simplicity, assume that logger keys are integers. Suppose that the dictionary is implemented as a linked list of key–logger pairs, whose nodes are represented in C as:

```
struct dnode { int key; struct logger* log;
                        struct dnode*  nxt; };
```

A typical representation predicate for such dictionaries is:

$$\begin{aligned} &\mathsf{Dict} : \mathsf{repr}\ (\mathsf{list}\ (\mathsf{int} \times \mathsf{logger})) \\ &\mathsf{Dict}\ []\ p \triangleq \lfloor p = \mathtt{null} \rfloor \\ &\mathsf{Dict}\ ((k, g) :: d)\ p \triangleq p \mapsto k \\ &\qquad\qquad \star\ \exists\, q_1.\, p+1 \mapsto q_1\ \star\ \mathsf{Logger}\ g\ q_1 \\ &\qquad\qquad \star\ \exists\, q_2.\, p+2 \mapsto q_2\ \star\ \mathsf{Dict}\ d\ q_2 \end{aligned}$$

When the dictionary is non-empty, the head key k is stored at p, a pointer to the head logger g is stored at $p+1$, and a pointer to the rest of the dictionary d is stored at $p+2$.

Step 0. Abstract indirection and pointer arithmetic. The "log" and "nxt" fields store intermediate pointers rather than embedding the referenced objects in place. We express such indirection using the *indirection transformer* "*" (section 4.4), so the definition below desugars to the one above. Changes are marked using the "just changed" color convention from fig. 3.

$$\begin{aligned} &\mathsf{Dict} : \mathsf{repr}\ (\mathsf{list}\ (\mathsf{int} \times \mathsf{logger})) \\ &\mathsf{Dict}\ []\ p \triangleq \lfloor p = \mathtt{null} \rfloor \\ &\mathsf{Dict}\ ((k, g) :: d)\ p \triangleq \\ &\qquad\qquad p \mapsto k \star {}^{*}\mathsf{Logger}\ g\ (p+1) \star {}^{*}\mathsf{Dict}\ d\ (p+2) \end{aligned}$$

We also use records (section 5.1.2) to syntactically encapsulate pointer arithmetic, allowing Dict to be further rewritten as below. Here, Cell is the points-to predicate asserting a value occupies a single cell:

$$\forall\, v\ p.\ \mathsf{Cell}\ v\ p \triangleq p \mapsto v$$

$$\begin{aligned} &\mathsf{Dict} : \mathsf{repr}\ (\mathsf{list}\ (\mathsf{int} \times \mathsf{logger})) \\ &\mathsf{Dict}\ []\ p \triangleq \lfloor p = \mathtt{null} \rfloor \\ &\mathsf{Dict}\ ((k, g) :: d)\ p \triangleq \\ &\qquad \mathsf{Record}\ \langle\, \mathsf{key}\colon \mathsf{Cell}\ k;\ \mathsf{log}\colon {}^{*}\mathsf{Logger}\ g;\ \mathsf{nxt}\colon {}^{*}\mathsf{Dict}\ d\, \rangle\ p \end{aligned}$$

When the dictionary is non-empty, p points to a record with three fields: the "key" field stores k within a single memory cell described by Cell; the "log" and "nxt" fields store g and d indirectly, that is, store an intermediate pointer to the corresponding objects.

Step 1. Identify borrowable subparts and redefine the container's representation predicate to track their borrowing states. We first identify the borrowable subparts, in this case the loggers, and lift their logical models (of type logger) to the borrow-aware form (of type ${}^{\flat}$logger), where each logger is annotated with its borrowing state. We use v', R' and f' to denote the borrow-aware forms of the logical value v, predicate R and function f, respectively. We then define a borrow-aware representation predicate $\mathsf{Dict}_{\mathsf{e}}$ to describe a dictionary whose elements are borrowable:

$$\begin{aligned} &\mathsf{Dict}_{\mathsf{e}} : \mathsf{repr}\ (\mathsf{list}\ (\mathsf{int} \times {}^{\flat}\mathsf{logger})) \\ &\mathsf{Dict}_{\mathsf{e}}\ []\ p \triangleq \lfloor p = \mathtt{null} \rfloor \\ &\mathsf{Dict}_{\mathsf{e}}\ ((k, g') :: d)\ p \triangleq \\ &\qquad \mathsf{Record}\ \langle\, \mathsf{key}\colon \mathsf{Cell}\ k;\ \mathsf{log}\colon {}^{*\flat}\mathsf{Logger}\ g';\ \mathsf{nxt}\colon {}^{*}\mathsf{Dict}_{\mathsf{e}}\ d\, \rangle\ p \end{aligned}$$

This definition states that when the dictionary is non-empty, the "log" field stores a borrowable logger g' indirectly.

Step 2. Enrich API specifications to reflect the validity of exposed internal pointers. To capture the fact that `set_level_all` updates loggers in place, we refine its specification to explicitly ensure pointer stability:

$$\begin{aligned} &\{\, \lfloor \forall\, g' \in d.\ \mathsf{isAvailable}\ g' \rfloor \star \mathsf{Dict}_{\mathsf{e}}\ d\ p \,\} \\ &\qquad \mathtt{set\_level\_all}(p, v) \\ &\{\, \lambda\ \_.\ \mathsf{Dict}_{\mathsf{e}}\ (\mathsf{map}\ (\lambda\ g'.\ g'[\![\mathsf{lvl} := v]\!])\ d)\ p \,\} \end{aligned}$$

isAvailable g' asserts that the underlying value of g' is accessible, i.e., g' is in the Owned or Offered state. This specification requires the availability of all loggers in the dictionary, allowing them to be offered at exposed locations. In the postcondition, $g'[\![\mathsf{lvl} := v]\!]$ denotes a borrow-aware substitution that preserves exposed locations.

We can now specify `get` to track the returned pointer in the borrowing state without detaching g ($[r]$ denotes $r :: []$):

$$\begin{aligned} &\textit{(weakest)}\ \{\, \lfloor d[k] = \mathsf{Owned}\ g \rfloor \star \mathsf{Dict}_{\mathsf{e}}\ d\ p \,\} \\ &\qquad \mathtt{get}(p, k)\ \{\, \lambda\ r.\ \mathsf{Dict}_{\mathsf{e}}\ d[k := \mathsf{Offered}\ g\ [r]]\ p \,\} \end{aligned}$$

This first specification allows loggers other than $d[k]$ to be unavailable; but it prohibits repeated calls to get with the same key, as it requires $d[k]$ to be in the Owned state. We can strengthen it to allow the queried logger to be either Owned or Offered:

(weaker) $\{ \lfloor d[k] = g' \wedge \mathsf{isAvailable}\ g' \rfloor \star \mathsf{Dict_e}\ d\ p \}$
$\mathtt{get}(p,\ k)\ \{\lambda\ r.\ \mathsf{Dict_e}\ d[k := \mathsf{pin}\ g'\ r]\ p\ \}$

Function pin $v'\ r$ pins a borrowable value v' at r. The predicate transformer "$^\flat$" ensures that if v' is already pinned, the existing and new locations coincide.

v'	pin $v'\ r$
Owned v	Offered v $(r :: [])$
Offered v $(q :: qs)$	Offered v $(r :: q :: qs)$
Borrowed $(q :: qs)$	Borrowed $(r :: q :: qs)$
Missing	Borrowed $(r :: [])$

We can further strengthen the specification by allowing the queried logger to be Borrowed or Missing as well:

(strongest) $\{ \lfloor d[k] = g' \rfloor \star \mathsf{Dict_e}\ d\ p \}$
$\mathtt{get}(p,\ k)\ \{\lambda\ r.\ \mathsf{Dict_e}\ d[k := \mathsf{pin}\ g'\ r]\ p\ \}$

Step 3. Prove correctness with explicit logical borrow and return steps around subpart access. Figure 3 illustrates a correctness proof of the logger example, with the correctness theorem stated as:

$(\forall\ g' \in d.\ \mathsf{isAvailable}\ g') \wedge (g'_k = d[k]) \rightarrow$
$\mathsf{let}\ d_1 \triangleq \mathsf{map}\ (\lambda\ g'.\ g'[\![\mathsf{lvl} := \mathtt{INFO}]\!])\ d\ \mathsf{in}$
$\{\ \mathsf{Dict_e}\ d\ \mathit{registry}\ \}$ (the logger program)
$\{\ \lambda\ \_.\ \mathsf{Dict_e}\ d_1[k := \mathsf{pin}\ g'_k[\![\mathsf{lvl} := \mathtt{DEBUG}]\!]\ \mathit{lgr}]\ \mathit{registry}\ \}$

Immediately before set_level, we perform logical borrow to detach the target logger's ownership from the dictionary. From the dictionary's perspective, the logger transitions from being offered at *lgr* (available for future borrowing) to being borrowed at *lgr* (held by another party), with *lgr* enabling future return.

After performing set_level, whose specification frames out the partial dictionary, we perform a logical-return step (using the handle *lgr*) to restore the updated logger's ownership to the dictionary, thereby recovering the full dictionary representation for subsequent operations.

4.3 Borrowability

In flat containers such as arrays and records, borrowable subparts are typically the stored items (elements), such as record fields and array entries. They can be individually manipulated without affecting the overall structure. In recursive containers such as lists and trees, borrowable subparts include both elements *and* recursively defined substructures, such as list tails and subtrees.

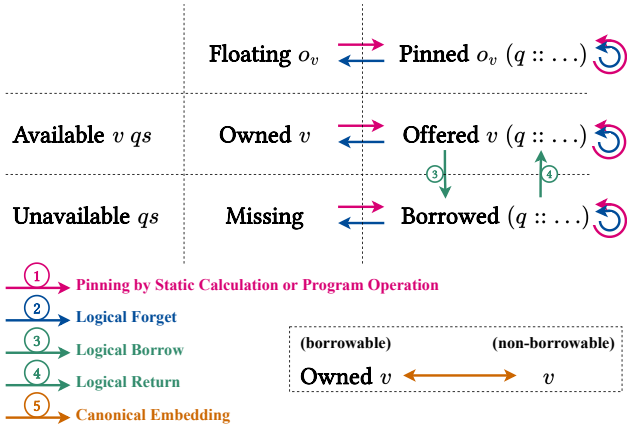

Figure 5. Borrowing states and transitions.

4.3.1 Borrowing States.

The logical datatype borrowable models the borrowing states and lifts a base type A to its borrow-aware form:

borrowable A (written $^\flat A$) $\triangleq$ (option A) $\times$ (list loc)

	Floating o_v $\triangleq (o_v,\ [])$	Pinned o_v $(q :: qs)$ $\triangleq (o_v,\ q :: qs)$
Available v qs $\triangleq$ (Some v, qs)	Owned v $\triangleq$ (Some v, [])	Offered v $(q :: qs)$ $\triangleq$ (Some v, $q :: qs$)
Unavailable qs $\triangleq$ (None, ps)	Missing $\triangleq$ (None, [])	Borrowed $(q :: qs)$ $\triangleq$ (None, $q :: qs$)

The Offered and Borrowed states capture the fact that the borrowable subpart remains at its previously exposed location q (with additional aliases qs); we call them the *pinned states*. Pinned states indicate the validity of exposed internal pointers to the element.

Owned and Missing are *floating states*. For flat containers such as arrays, vectors and records, floating states can be upgraded to pinned states purely by static calculation (transition ① in fig. 5), i.e., by computing each subpart's location from the container's base address and the subpart's offset that is known statically.

Missing and Borrowed are *unavailable states*, indicating that the subpart has been logically detached from the container and replaced by a tagged hole. In contrast, *available states*, Owned and Offered, indicate that the container owner has ownership of this subpart.

4.3.2 Containers with Borrowable Subparts.

Logical datatypes. Consider the following standard definition of a logical list. We write [] for Nil, $x :: l$ for Cons $x\ l$, and $[x_1; \ldots; x_n]$ for Cons x_1 $(\ldots (\mathsf{Cons}\ x_n\ \mathsf{Nil}))$.

$$\mathsf{list}\ A \triangleq \mathsf{Nil} \mid \mathsf{Cons}\ (x : A)\ (l : \mathsf{list}\ A)$$

By lifting the element type from A to ${}^\flat A$, we obtain lists whose elements can be borrowed independently:

$$\mathsf{list_e}\ A \triangleq \mathsf{list}\ {}^\flat A$$

This definition captures the borrowing pattern in which each element of a container can be accessed independently. For instance, in the logger example (section 4.2), the borrow-aware dictionary has type list $(\mathsf{int} \times {}^\flat\mathsf{logger})$.

Supporting borrowable recursive substructures (such as list tails or subtrees) requires modifying the inductive definition of the container (its underlying "shape"). For example, $\mathsf{list_s}$ describes lists in which each tail is borrowable:

$$\mathsf{list_s}\ A \triangleq \mathsf{Nil_s} \mid \mathsf{Cons_s}\ (x : A)\ (l' : {}^\flat(\mathsf{list_s}\ A))$$

We write $\overline{[]}$ for $\mathsf{Nil_s}$ and $x :: l'$ for $\mathsf{Cons_s}\ x\ l'$.

Combining these two forms, we obtain $\mathsf{list_{es}}\ A$ for lists in which both elements and tails are borrowable:

$$\mathsf{list_{es}}\ A \triangleq \mathsf{list_s}\ {}^\flat A$$

Representation predicates. The generic *borrow transformer* on representation predicates, liftToB, lifts a predicate R of type repr A to a predicate of type repr ${}^\flat A$. We write ${}^\flat R$ for liftToB R. The transformer is defined as follows:

$${}^\flat R\ (o_v, qs)\ p \triangleq {}_{\mathsf{val}}^{\flat}R\ o_v\ p \star {}_{\mathsf{loc}}^{\flat}R\ qs\ p$$

$${}_{\mathsf{val}}^{\flat}R\ (\mathsf{Some}\ v)\ p \triangleq R\ v\ p \qquad {}_{\mathsf{loc}}^{\flat}R\ qs\ p \triangleq \lfloor \forall\ q \in qs.\ q = p \rfloor$$

$${}_{\mathsf{val}}^{\flat}R\ (\mathsf{None}\ )\ p \triangleq \lfloor \top \rfloor$$

The representation predicate ${}^\flat R$ for the borrowable value (o_v, qs) has two components: the value part ${}_{\mathsf{val}}^{\flat}R\ o_v\ p$, which describes the underlying value if available; and the location part ${}_{\mathsf{loc}}^{\flat}R\ qs\ p$, which asserts that every pointer in the exposed alias list qs is equal to the canonical location p.

Tracking an alias list rather than a single optional exposed pointer simplifies specifications: equality among all aliases is encapsulated in ${}^\flat R$, so functions that read or create aliases do not need to track it explicitly. For example, consider `get` on a dictionary where the queried logger g already has an exposed pointer q. One could explicitly track the new alias r:

$$\{\ \lfloor d[k] = \mathsf{Offered}\ g\ [q] \rfloor \star \mathsf{Dict_e}\ d\ p\ \}$$
$$\mathtt{get}(p, k)\ \{\ \lambda\ r.\ \lfloor r = q \rfloor \star \mathsf{Dict_e}\ d\ p\ \}$$

Using the alias list, the same postcondition can be expressed equivalently and more cleanly as (in the same manner as the specifications in section 4.2):

$$\{\ \lambda\ r.\ \mathsf{Dict_e}\ d[k := \mathsf{Offered}\ g\ [r; q]]\ p\ \}$$
$$\equiv \{\ \lambda\ r.\ \mathsf{Dict_e}\ d[k := \mathsf{pin}\ d[k]\ r]\ p\ \}$$

The borrow transformer "${}^\flat$" enables modular lifting of representation predicates to support borrow tracking. Starting from the standard list representation predicate List (section 1), we define $\mathsf{List_e}$ that supports borrowing elements:

$$\mathsf{List_e} : (\mathsf{repr}\ A) \to \mathsf{repr}\ (\mathsf{list_e}\ A) \quad \mathsf{List_e}\ R \triangleq \mathsf{List}\ {}^\flat R$$

A second variant, $\mathsf{List_s}$, supports borrowing tails:

$$\mathsf{List_s} : (\mathsf{repr}\ A) \to \mathsf{repr}\ (\mathsf{list_s}\ A)$$

$$\mathsf{List_s}\ R\ \overline{[]}\ p \triangleq \lfloor p = \mathtt{null} \rfloor$$

$$\mathsf{List_s}\ R\ (x :: l')\ p \triangleq R\ x\ p \star \exists\, q.\ p + 1 \mapsto q \star {}^\flat(\mathsf{List_s}\ R)\ l'\ q$$

Similarly, $\mathsf{List_{es}}$ combines both forms, supporting borrowable elements and borrowable tails:

$$\mathsf{List_{es}} : (\mathsf{repr}\ A) \to \mathsf{repr}\ (\mathsf{list_{es}}\ A) \quad \mathsf{List_{es}}\ R \triangleq \mathsf{List_s}\ {}^\flat R$$

4.3.3 Borrowing State Transitions.

Figure 5 shows the transitions between the borrowing states.

Transition ① (Pinning by Static Calculation or Program Operation) pins a container subpart at a newly exposed location. Static calculation, as already discussed in section 4.3.1, derives the subpart's location based on static knowledge. It is applicable only in flat containers.

For most pointer-based containers, subpart locations are often not statically known, and must instead be computed dynamically by executing traversal operations. For example, in the logger example, the `get` function exposes the location of a logger stored in a linked list, whose address cannot be determined statically.

A pinning transition always produces a pinned state. Assume that the newly exposed location is r. Such a transition can convert a floating state into a pinned state, for example, from Owned v to Offered v $[r]$. If starting from an already pinned state such as Offered v $(q :: qs)$, it extends the alias list with r, yielding Offered v $(r :: q :: qs)$.

Transition ② (Logical Forget) is the reverse of pinning: it logically invalidates an exposed pointer by discarding the pure constraint that tracks it.

$$\textit{(logical forget)} \quad {}^\flat R\ (o_v, q :: qs)\ p \vdash {}^\flat R\ (o_v, qs)\ p$$

Logical forget always starts with a pinned state. If all pointers in the alias list are forgotten, it yields a floating state; otherwise the produced state is still pinned. Logical return is useful for unifying a state with one without certain aliases, such as postconditions that do not track pointers exposed temporarily within the function body.

Transition ③ (Logical Borrow) and ***Transition ④ (Logical Return)*** implement the key feature of our model: supporting flexible ownership transfers.

Specifically, a subpart offered at an exposed location can be logically borrowed from the container for local reasoning. Its ownership can be later reclaimed by the original container, as long as the subpart is not moved (but it may be mutated). Formally, the logical borrow and return rules are captured by the following equivalence:

$${}^\flat R\,(\mathsf{Offered}\ v\ (q :: qs))\ p \equiv {}^\flat R\,(\mathsf{Borrowed}\ (q :: qs))\ p \star R\ v\ q$$

Figure 2 provides an intuitive illustration. A container c with elements e_1 offered at p_1 and e_3 offered at p_3 can be decomposed into (1) a partial container with holes at p_1 and p_3, and (2) the two detached elements e_1 and e_3. This

decomposition corresponds to logical borrow, and the reverse operation (recomposition) to logical return.

Transition ⑤ (Canonical Embedding) shows that non-borrowable values are equivalent to owned borrowable value:

$$R\ v\ p \equiv {}^{\flat}R\ (\mathsf{Owned}\ v)\ p$$

We often omit this transition in the rest of the paper.

4.4 Indirection

Many containers store their elements or substructures *indirectly* through intermediate pointers rather than embedding them in place. Such indirection is standard in pointer-based containers such as linked lists and trees.

Unlike borrowability, modeling indirection does not require modifying the logical datatype, as functional datatypes already abstract away low-level pointer indirection.

We introduce the *indirection transformer* "$*$" to map a representation predicate R for values stored in place to a predicate for values stored via an intermediate pointer. Although "$*$" is not directly related to borrowing, it composes nicely with the borrow transformer "$\flat$".

$$^{*} : \forall\ A.\ (\mathsf{repr}\ A) \to \mathsf{repr}\ A \quad {}^{*}R\ v\ p \triangleq \exists\, q.\, p \mapsto q \star R\ v\ q$$

With "$*$", the non-empty case of List can be written as:

$$\mathsf{List}\ R\ (x :: l)\ p \equiv R\ x\ p \star {}^{*}(\mathsf{List}\ R)\ l\ (p+1)$$

4.5 Compositions of Predicate Transformers

The two representation predicate transformers "$\flat$" and "$*$" serve as the fundamental abstraction for modeling recursive containers with borrowable subparts. By composing them, we can express different patterns of memory organization. Here, "$\circ$" denotes the standard function composition operator: $(g \circ f)\ x = g\ (f\ x)$.

$^{*} \circ {}^{\flat}$ models a borrowable value stored indirectly. For example, the non-empty case of List_s can be written as:

$$\mathsf{List}_s\ (x :: l')\ p \equiv R\ x\ p \star {}^{*\flat}(\mathsf{List}_s\ R)\ l'\ (p+1)$$

Here, ${}^{*\flat}(\mathsf{List}_s\ R)\ l'\ (p+1)$ states that an intermediate pointer is stored at $p+1$, and this pointer points to the borrowable tail l'. This predicate is useful for reasoning about in-place updates to the tail, but not about modifying the intermediate pointer, since it is not borrowable.

${}^{\flat} \circ {}^{*}$ models the case where the intermediate pointer itself can be borrowed (along with the memory it points to). Consider the lists modeled by List'_s which is defined as follows:

$$\mathsf{List}'_s : (\mathsf{repr}\ A) \to \mathsf{repr}\ (\mathsf{list}_s\ A)$$
$$\mathsf{List}'_s\ R\ \overline{[]}\ p \triangleq \lfloor p = \mathtt{null} \rfloor$$
$$\mathsf{List}'_s\ R\ (x :: l')\ p \triangleq R\ x\ p \star {}^{\flat *}(\mathsf{List}'_s\ R)\ l'\ (p+1)$$

Here, ${}^{\flat *}(\mathsf{List}'_s\ R)\ l'\ (p+1)$ states that a borrowable intermediate pointer is stored at $p+1$, and borrowing this pointer transitively borrows the tail it points to. This predicate is useful when the client may update the tail by modifying the intermediate pointer, for example, to point to a different list.

We have ${}^{*\flat}R \neq {}^{\flat *}R$, ${}^{**}R \neq {}^{*}R$, but

$$^{\flat\flat}R\ (\mathsf{Some}\ v',\ qs)\ p = {}^{\flat}R\ v'\ p \star {}^{\flat\flat}_{\mathsf{loc}}R\ qs\ p$$
$$^{\flat\flat}R\ (\mathsf{None},\ qs)\ p = {}^{\flat\flat}_{\mathsf{loc}}R\ qs\ p$$

By composition of "$\flat$" and "$*$", we can express a wide range of memory manipulation patterns in a principled and reusable way. For example, ${}^{\flat *\flat}R$ is useful when the intermediate pointer and pointed-to object are both borrowable.

5 Case Studies

In this section, we illustrate logical pinning across a series of containers. We adopt a simple imperative λ-calculus with dynamic allocation, along with the linear separation logic presented in section 3.1. The language includes let-bindings, conditional expressions, fixpoints, equality checks, pointer arithmetic, and mutable arrays. Its core heap-manipulating primitives are as follows:

$\mathtt{alloc}(n)$	Allocate n contiguous uninitialized cells, and return the starting location.
$\mathtt{free}(n,\ p)$	Deallocate n contiguous cells starting at p.
$p \leftarrow v$	Write value v to memory location p.
$!p$	Read value at memory location p.

Our development, including correctness proofs of container implementations with respect to our specifications and case-study programs, is formalized on top of a variant of the core CFML calculus in the Rocq proof assistant. The formalization is available at:

https://github.com/epfl-systemf/logical-pinning

5.1 Flat Containers

5.1.1 Single Cells. We present two examples of cells that, while not useful in isolation as cells are the smallest memory units, become valuable if combined into larger structures.

A cell with a borrowable primitive value. Such a cell can be read from or written to when the borrowable value is available. Neither operations invalidate pointers to this cell:

$$\{\ {}^{\flat}\mathsf{Cell}\ (\mathsf{Available}\ v\ qs)\ p\ \}\ !p$$
$$\{\ \lambda\ r.\ \lfloor r = v \rfloor \star {}^{\flat}\mathsf{Cell}\ (\mathsf{Available}\ v\ qs)\ p\ \}$$
$$\{\ {}^{\flat}\mathsf{Cell}\ (\mathsf{Available}\ v\ qs)\ p\ \}\ p \leftarrow u$$
$$\{\ {}^{\flat}\mathsf{Cell}\ (\mathsf{Available}\ u\ qs)\ p\ \}$$

A cell with an indirectly-stored borrowable value. Assume that the cell at location p is described by ${}^{*}R\ v\ p$, that is, it stores an intermediate pointer to v, and v can be described by predicate R. Reads and writes to this cell operate on the intermediate pointer without affecting v:

$$\{\ {}^{*}R\ v\ p\ \}\ !p\ \{\ \lambda\ r.\ \mathsf{Cell}\ p\ r \star R\ v\ r\ \}$$
$$\{\ {}^{*}R\ v\ p\ \}\ p \leftarrow q\ \{\ \lambda\ \_.\ \mathsf{Cell}\ p\ q \star \exists\, q_v.\, R\ v\ q_v\ \}$$

Although the above specification is precise, it exposes the internal structure of "*" and thus breaks the abstraction of indirection, reducing modularity. For example, it does not allow reading from or writing to the same cell twice, since the pre- and postconditions do not have the same shape.

The intermediate pointer being operated on is an internal pointer of this single-cell container, thereby its exposure or mutation can instead be handled by our borrowing model:

$$\{\ {}^{*\flat}R\ v'\ p\ \}\ !p\ \{\ \lambda\ r.\ {}^{*\flat}R\ (\mathsf{pin}\ v'\ r)\ p\ \}$$
$$\equiv \{\ {}^{*\flat}R\ v'\ p\ \}\ !p\ \{\ \lambda\ r.\ {}^{*\flat}R\ (\mathsf{Borrowed}\ [r])\ p \star {}^{\flat}R\ v'\ r\ \}$$
$$\{\ {}^{*\flat}R\ (\mathsf{Borrowed}\ [r])\ p \star {}^{\flat}R\ v'\ r\ \}\ p \leftarrow q$$
$$\{\ \lambda\ \_.\ {}^{*\flat}R\ (\mathsf{Borrowed}\ [q])\ p \star {}^{\flat}R\ v'\ r\ \}$$

Here, we make the inner value borrowable. Exposing the intermediate pointer pins the inner value, and replacing this pointer with q is treated as replacing the inner value with a hole at q (the target of q is unknown).

5.1.2 Arrays and Records. A plain array is a contiguous block of memory without a header as in C, and we model it as a list of values laid out sequentially in memory:

$$\mathsf{Array} : \forall\ A.\ (\mathsf{repr}\ A) \to \mathsf{repr}\ (\mathsf{list}\ A)$$
$$\mathsf{Array}\ R\ []\ p \triangleq \lfloor p = \mathtt{null} \rfloor$$
$$\mathsf{Array}\ R\ (x :: l)\ p \triangleq \lfloor p \neq \mathtt{null} \rfloor \star R\ x\ p \star \mathsf{Array}\ R\ l\ (p + 1)$$

We consider records as *heterogeneous* arrays, where each field occupies one cell that stores a value in place or an intermediate pointer to the value, and the representation predicate of each field may differ. To uniformly describe such fields, we introduce a helper type ReprVal to package a value with its representation predicate:

$$\mathsf{ReprVal} \triangleq \mathsf{RV}\ (t : \mathsf{type})\ (R_t : \mathsf{repr}\ t)\ (v : t)$$

When t is clear from context, we write RV $R\ v$ with the type left implicit.

Each field label f is associated with a natural number index offset_f, indicating the field's offset from the entry location of the record. The representation predicate for records is defined as follows:

$$\mathsf{Record} : \mathsf{repr}\ (\mathsf{list}\ (\mathsf{Field} \times \mathsf{ReprVal}))$$
$$\mathsf{Record}\ []\ p \triangleq \lfloor p = \mathtt{null} \rfloor$$
$$\mathsf{Record}\ ((f,\ \mathsf{RV}\ R\ v) :: l)\ p \triangleq$$
$$\lfloor p \neq \mathtt{null} \rfloor \star R\ v\ (p + \mathit{offset}_f) \star \mathsf{Record}\ l\ p$$

We write $\langle\ f_0 : R_0\ v_0;\ \ldots;\ f_{n-1} : R_{n-1}\ v_{n-1}\ \rangle$ for a record with n fields f_0 to f_{n-1}. For each $i \in \{0, \ldots, n-1\}$, field f_i stores value v_i with representation predicate R_i.

Records serve as a fundamental abstraction for defining containers in our approach. Recall the list representation predicate List_s (section 4.3.2), its non-empty case can be encoded as a record with two fields:

$$\mathsf{List}_s\ R\ (x :: l)\ p \equiv \mathsf{Record}\ \langle\ \mathsf{elem} : R\ x;\ \mathsf{next} : {}^{*\flat}(\mathsf{List}\ R)\ l\ \rangle\ p$$

where $\mathit{offset}_{\mathsf{elem}}$ is 0, $\mathit{offset}_{\mathsf{next}}$ is 1.

We add the following notations as syntactic sugar for field access and exposing field location:

Notation	C Syntax	Desugars to
$p.f$	`(*p).f`, `p→f`	$!(p + \mathit{offset}_f)$
$p.f \leftarrow v$	`(*p).f = v`, `p→f = v`	$(p + \mathit{offset}_f) \leftarrow v$
$\&p.f$	`&(p→f)`	$p + \mathit{offset}_f$

Field access. Accessing a record field reduces to accessing a single cell (section 5.1.1), hence we can lift the single-cell specifications for field operations. For example, reading a field that stores a borrowable value in place is specified as follows. Crucially, only the target field's value must be available, and other fields may be borrowed independently.

$$\{\ \lfloor l[f] = \mathsf{RV}\ {}^{\flat}\mathsf{Cell}\ (\mathsf{Available}\ v\ qs) \rfloor \star \mathsf{Record}\ l\ p\ \}$$
$$p.f\ \{\ \lambda\ r.\ \lfloor r = v \rfloor \star \mathsf{Record}\ l\ p\ \}$$

Exposing field location. Because records are flat containers, a field can always be pinned at its address statically:

$$\mathsf{Record}\ l[f := \mathsf{RV}\ {}^{\flat}R\ v']\ p$$
$$\equiv \mathsf{Record}\ l[f := \mathsf{RV}\ {}^{\flat}R\ (\mathsf{pin}\ v'\ (p + \mathit{offset}_f))]\ p$$

Alternatively, the location can be obtained dynamically:

$$\{\ \lfloor l[f] = \mathsf{RV}\ {}^{\flat}R\ v' \rfloor \star \mathsf{Record}\ l\ p\ \}\ \&p.f$$
$$\{\ \lambda\ r.\ \mathsf{Record}\ l[f := \mathsf{RV}\ {}^{\flat}R\ (\mathsf{pin}\ v'\ r)]\ p\ \}$$

By the definition of Record, r equals $p + \mathit{offset}_f$.

5.2 Recursive Containers

Recursive containers are the natural use case of the logical-pinning model: element locations are not statically known, and their pointer-rich APIs often expose internal pointers. Our model marks these pointers on demand; we demonstrate its generality and practicality on linked lists and trees, by deriving precise specifications for standard APIs.

5.2.1 Linked Lists. We study singly linked lists, with nodes represented in C as `lnode` (sections 1 and 4.3.2). We start with the traditional representation of lists — a logical model whose elements and tails are not borrowable:

$$\mathsf{CList} : \mathsf{repr}\ (\mathsf{list}\ \mathsf{int}) \qquad \mathsf{CList} \triangleq \mathsf{List}\ \mathsf{Cell}$$

Consider the specification of the function `size`, which computes the list's length $|l|$, in this model:

$$\{\ \mathsf{CList}\ l\ p\ \}\ \mathtt{size}(p)\ \{\ \lambda\ r.\ \lfloor r = |l| \rfloor \star \mathsf{CList}\ l\ p\ \}$$

Even for such a simple function, the CList-base specification is imprecise. `size` neither exposes internal pointers, nor relocates objects, nor transfers ownership, yet (1) it demands more permissions than necessary, as `size` depends only on the list shape and does not need ownership of elements; and (2) it does not capture pointer stability, which makes pointer tracking impossible.

Let us instead use logical pinning. We introduce CList_e, which describes the same list but with borrowable elements.

$$
\begin{array}{rcl}
\{\ \mathsf{CList_e}\ l\ p\ \} & \texttt{size}(p) & \{\ \lambda\, r.\ \lfloor r = |l| \rfloor \star \mathsf{CList_e}\ l\ p\ \} \\
\{\ \mathsf{CList_e}\ l\ p\ \} & \texttt{is\_empty}(p) & \{\ \lambda\, r.\ \lfloor r = \mathsf{true} \leftrightarrow l = [] \rfloor \star \mathsf{CList_e}\ l\ p\ \} \\
\{\ \mathsf{CList_e}\ ((\mathsf{Available}\ v\ qs) :: l)\ p\ \} & p.\mathsf{elem} & \{\ \lambda\, r.\ \lfloor r = v \rfloor \star \mathsf{CList_e}\ ((\mathsf{Available}\ v\ qs) :: l)\ p\ \} \\
\{\ \mathsf{CList_e}\ ((\mathsf{Available}\ v\ qs) :: l)\ p\ \} & p.\mathsf{elem} \leftarrow u & \{\ \lambda\ \_.\ \mathsf{CList_e}\ ((\mathsf{Available}\ u\ qs) :: l)\ p\ \} \\
\{\ \lfloor \forall\, v' \in l.\ \mathsf{isAvailable}\ v' \rfloor \star \mathsf{CList_e}\ l\ p\ \} & \texttt{incr\_all}(p) & \{\ \lambda\, r.\ \mathsf{CList_e}\ (\mathsf{map_e}\ (\lambda\, v.\ v + 1)\ l)\ p\ \} \\
\{\ \lfloor n < |l| \wedge \mathsf{nth}\ n\ l = \mathsf{Available}\ v\ qs \rfloor \star \mathsf{CList_e}\ l\ p\ \} & \texttt{nth\_elem}(n, p) & \{\ \lambda\, r.\ \lfloor r = v \rfloor \star \mathsf{CList_e}\ l\ p\ \} \\
\{\ \lfloor n < |l| \rfloor \star \mathsf{CList_e}\ l\ p\ \} & \texttt{nth\_elem\_ptr}(n, p) & \{\ \lambda\, r.\ \mathsf{CList_e}\ l[n := \mathsf{pin}\ (\mathsf{nth}\ n\ l)\ r]\ p\ \} \\
\{\ \mathsf{CList_e}\ (v' :: l)\ p\ \} & \texttt{pop\_front}(p) & \{\ \lambda\, r.\ \mathsf{CList_e}\ l\ r \star {}^{\flat}\mathsf{Cell}\ v'\ p\ \}
\end{array}
$$

Figure 6. API specification for lists with borrowable elements and non-borrowable tails.

$$
\begin{array}{rcl}
\{\ \lfloor \mathsf{allTailAvl}\ (\mathsf{Owned}\ l) \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} & \texttt{size}(p) & \{\ \lambda\, r.\ \lfloor r = |l| \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} \\
\{\ \mathsf{CList_{es}}\ l\ p\ \} & \texttt{is\_empty}(p) & \{\ \lambda\, r.\ \lfloor r = \mathsf{true} \leftrightarrow l = \overline{[]} \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} \\
\{\ \mathsf{CList_{es}}\ ((\mathsf{Available}\ v\ qs) \mathbin{\overline{::}} l')\ p\ \} & p.\mathsf{elem} & \{\ \lambda\, r.\ \lfloor r = v \rfloor \star \mathsf{CList_{es}}\ ((\mathsf{Available}\ v\ qs) \mathbin{\overline{::}} l')\ p\ \} \\
\{\ \mathsf{CList_{es}}\ ((\mathsf{Available}\ v\ qs) \mathbin{\overline{::}} l')\ p\ \} & p.\mathsf{elem} \leftarrow u & \{\ \lambda\ \_.\ \mathsf{CList_{es}}\ ((\mathsf{Available}\ u\ qs) \mathbin{\overline{::}} l')\ p\ \} \\
\{\ \lfloor \mathsf{allAvl}\ (\mathsf{Owned}\ l) \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} & \texttt{incr\_all}(p) & \{\ \lambda\, r.\ \mathsf{CList_{es}}\ (\mathsf{map_{es}}\ (\lambda\, v.\ v + 1)\ l)\ p\ \} \\
\left\{ \begin{array}{r} \lfloor n < |l| \wedge \mathsf{nth_{es}}\ n\ l = \mathsf{Available}\ v\ qs \rfloor \\ \star\, \mathsf{CList_{es}}\ l\ p \end{array} \right\} & \texttt{nth\_elem}(n, p) & \{\ \lambda\, r.\ \lfloor r = v \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} \\
\{\ \lfloor n < |l| \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} & \texttt{nth\_elem\_ptr}(n, p) & \{\ \lambda\, r.\ \mathsf{CList_{es}}\ l[n := \mathsf{pin}\ (\mathsf{nth_{es}}\ n\ l)\ r]\ p\ \} \\
\{\ \mathsf{CList_{es}}\ (v' \mathbin{\overline{::}} \mathsf{Available}\ l\ qs)\ p\ \} & \texttt{pop\_front}(p) & \{\ \lambda\, r.\ \mathsf{CList_{es}}\ l\ r \star \lfloor \forall\, q \in qs.\ q = r \rfloor \star {}^{\flat}\mathsf{Cell}\ v'\ p\ \} \\
\{\ \mathsf{CList_{es}}\ (v' \mathbin{\overline{::}} l')\ p\ \} & p.\mathsf{next} & \{\ \lambda\, r.\ \mathsf{CList_{es}}\ (v' \mathbin{\overline{::}} \mathsf{pin}\ l'\ r)\ p\ \} \\
\{\ \mathsf{CList_{es}}\ (v' \mathbin{\overline{::}} l')\ p\ \} & p.\mathsf{next} \leftarrow q & \{\ \lambda\, r.\ \mathsf{CList_{es}}\ (v' \mathbin{\overline{::}} \mathsf{Borrowed}\ [q])\ p \star \exists\, q_{l'}.\ {}^{\flat}\mathsf{CList_{es}}\ l'\ q_{l'}\ \} \\
\{\ \lfloor n < |l| \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} & \texttt{nth\_tail}(n, p) & \{\ \lambda\, r.\ \mathsf{CList_{es}}\ (\mathsf{substTail_{es}}\ n\ (\mathsf{pin}\ (\mathsf{skipn_{es}}\ n\ l)\ r)\ l)\ p\ \} \\
\{\ \mathsf{CList_{es}}\ l\ p\ \} & \texttt{push\_front}(v, p) & \{\ \lambda\, r.\ \mathsf{CList_{es}}\ ((\mathsf{Owned}\ v) \mathbin{\overline{::}} \mathsf{Offered}\ l\ [p])\ r\ \} \\
\left\{ \begin{array}{r} \lfloor \mathsf{allTailAvl}\ (\mathsf{Owned}\ l_1) \rfloor \\ \star\, \mathsf{CList_{es}}\ l_1\ p_1 \star \mathsf{CList_{es}}\ l_2\ p_2 \end{array} \right\} & \texttt{append}(p_1, p_2) & \{\ \lambda\, r.\ {}^{\flat}\mathsf{CList_{es}}\ ((\mathsf{Offered}\ l_1\ [p_1]) \mathbin{\overline{++}} (\mathsf{Offered}\ l_2\ [p_2]))\ r\ \} \\
\{\ \lfloor \mathsf{allTailAvl}\ (\mathsf{Owned}\ l) \rfloor \star \mathsf{CList_{es}}\ l\ p\ \} & \texttt{push\_back}(v, p) & \{\ \lambda\, r.\ {}^{\flat}\mathsf{CList_{es}}\ ((\mathsf{Offered}\ l\ [p]) \mathbin{\overline{++}} [\mathsf{Owned}\ v])\ r\ \}
\end{array}
$$

Figure 7. API specification for lists with borrowable elements and borrowable tails. allTailAvl l' (resp. allAvl l') asserts every tail (resp. subpart) of l' is available; $l'_1 \mathbin{\overline{++}} l'_2$ denotes the borrow-aware concatenation of two lists; $[x]$ denotes $x \mathbin{\overline{::}} \mathsf{Owned}\ \overline{[]}$.

This lets specifications require only the necessary element ownership, and track pointers to elements:

$$\mathsf{CList_e} : \mathsf{repr}\ (\mathsf{list}\ {}^{\flat}\mathsf{int}) \qquad \mathsf{CList_e} \triangleq \mathsf{List}\ {}^{\flat}\mathsf{Cell}$$

Figure 6 shows a complete $\mathsf{CList_e}$-based API specification. In particular, the new specification of `size` permits elements to be unavailable, preserves their borrowing states, and therefore preserves all exposed pointers tracked in those states. Other functions are `is_empty`, which returns true iff the list is empty; $p.\mathsf{elem}$ and $p.\mathsf{elem} \leftarrow u$, which read and write the "elem" field; `incr_all`, which increments all elements in place; `nth_elem` and `nth_elem_ptr`, which return the value of, and a pointer to, the n-th element; and `pop_front`, which removes the head and returns a pointer to the new head.

These $\mathsf{CList_e}$-based specifications are more precise than CList-based ones: they require minimal element ownership, and account for all possible exposed pointers to elements. For example, the specification of `nth_elem_ptr` tracks the returned element pointer by simply pinning it.

Some APIs also expose pointers to tails, rather than just elements. For those, we introduce $\mathsf{CList_{es}}$, which makes both elements and tails borrowable:

$$\mathsf{CList_{es}} : \mathsf{repr}\ (\mathsf{list_s}\ {}^{\flat}\mathsf{int}) \qquad \mathsf{CList_{es}} \triangleq \mathsf{List_s}\ {}^{\flat}\mathsf{Cell}$$

Figure 7 shows the $\mathsf{CList_{es}}$-based API, with operations that expose tail pointers highlighted. Such operations are difficult to specify precisely with $\mathsf{CList_e}$, and therefore do not appear in fig. 6. They include operations on the `next` field, `push_front` (which adds a new head), `append` (which concatenates a second list), and `push_back` (which appends a new node at the tail).

These $\mathsf{CList_{es}}$-based specifications are *maximally* precise: they account for every internal pointer to elements and tails, and require minimal ownership. For example, the specification of `append` preserves the two original head pointers p_1 and p_2 in a clean and natural form. All $\mathsf{CList_e}$-based specifications can be derived from the $\mathsf{CList_{es}}$-based ones.

Case study: swaps. Our specification of `nth_elem_ptr` requires no element ownership, and tracks exposed pointers without altering the container representation. This lets us prove the correctness of an element-swapping program that invokes `nth_elem_ptr` twice, on the same or distinct

indices, to obtain element pointers, and then calls `memswap` to exchange their contents (section B). Here too, we see that logical pinning naturally supports traditionally hard-to-verify patterns.

5.2.2 Binary Trees. We define $\mathsf{tree_s}$ with predicate $\mathsf{Tree_s}$ to model binary trees with borrowable subtrees:

$$\mathsf{tree_s}\ A \triangleq \mathsf{Leaf_s} \mid \mathsf{Node_s}\ (x : A)\ (t_l', t_r' : {}^{\flat}(\mathsf{tree_s}\ A))$$
$$\mathsf{Tree_s} : (\mathsf{repr}\ A) \to \mathsf{repr}\ (\mathsf{tree_s}\ A)$$
$$\mathsf{Tree_s}\ R\ \mathsf{Leaf_s}\ p \triangleq \lfloor p = \mathtt{null} \rfloor$$
$$\mathsf{Tree_s}\ R\ (\mathsf{Node_s}\ x\ t_l'\ t_r')\ p \triangleq \mathsf{Record}$$
$$\langle\, \mathsf{elem}\colon R\ x;\ \mathsf{lt}\colon {}^{*\flat}(\mathsf{Tree_s}\ R)\ t_l';\ \mathsf{rt}\colon {}^{*\flat}(\mathsf{Tree_s}\ R)\ t_r' \,\rangle\ p$$

In contrast, a custom representation predicate $\mathsf{Tree_{hc}}$ for trees with holes and cuts (section 3.3) would be written as:

$$\mathsf{Tree_{hc}} : (\mathsf{repr}\ A) \to \mathsf{repr}\ (\mathsf{tree_{hc}}\ A)$$
$$\mathsf{Tree_{hc}}\ R\ \mathsf{Leaf}\ p \triangleq \lfloor p = \mathtt{null} \rfloor$$
$$\mathsf{Tree_{hc}}\ R\ (\mathsf{Node}\ x\ t_l\ t_r)\ p \triangleq \mathsf{Record}$$
$$\langle\, \mathsf{elem}\colon R\ x;\quad \mathsf{lt}\colon {}^{*}(\mathsf{Tree_{hc}}\ R)\ t_l';\ \mathsf{rt}\colon {}^{*}(\mathsf{Tree_{hc}}\ R)\ t_r' \,\rangle\ p$$
$$\mathsf{Tree_{hc}}\ R\ (\mathsf{Hole}\ q\ t_l\ t_r)\ p \triangleq \mathsf{Record}$$
$$\langle\, \mathsf{elem}\colon (= p)\ x;\ \mathsf{lt}\colon {}^{*}(\mathsf{Tree_{hc}}\ R)\ t_l';\ \mathsf{rt}\colon {}^{*}(\mathsf{Tree_{hc}}\ R)\ t_r' \,\rangle\ p$$
$$\mathsf{Tree_{hc}}\ R\ (\mathsf{Cut}\ q)\ p \triangleq \lfloor p = q \rfloor$$

Our model subsumes this technique: our type $\mathsf{tree_s}\ {}^{\flat}A$ describes trees with *both* borrowable elements and subtrees, unifying holes and cuts as just Borrowed. This lets us reproduce the motivating example of the original paper on trees with holes and cuts [9], a `find` function that returns a subtree at a given path, with an enhanced specification that supports repeated lookups and guarantees pointer stability:

$$\{\, {}^{\flat}(\mathsf{Tree_s}\ R)\ t'\ p \star \lfloor \mathsf{path}\ i\ t'\ t_1' \rfloor \,\}\ \mathtt{find}(i, p)$$
$$\{\, \lambda\ r.\ {}^{\flat}(\mathsf{Tree_s}\ R)\ t'[\![i := \mathsf{pin}\ t_1'\ r]\!]\ p \,\}$$

Here, $\mathsf{path}\ i\ t'\ t_1'$ states that a valid path i in tree t' reaches a subtree t_1', and $t'[\![i := t_x']\!]$ denotes the borrow-aware substitution of t' in which the subtree at i is replaced by t_x'.

Case study: binary-tree left rotation. In this case study, we show that our model simplifies some complex proofs and enables more precise specifications.

During a binary-tree left rotation (fig. 1), a subtree is temporarily shared between two trees. In the traditional approach, this situation requires unfolding the tree abstraction two layers deep and managing the resulting split nodes and subtrees, which is conceptually cumbersome. An intermediate state has the following form, where Tree is a traditional representation predicate of binary trees:

$$\left\{ \begin{array}{l} p \mapsto x \ \star\ {}^{*}(\mathsf{Tree}\ R)\ t_l\ (p+1) \ \star\ p+2 \mapsto p_r \\ \star\ p_r \mapsto x_r \ \star\ p_r + 1 \mapsto p_{rl} \ \star\ {}^{*}(\mathsf{Tree}\ R)\ t_{rr}\ (p_r + 2) \\ \star\ \lfloor \forall\ q \in qs_r.\ q = p_r \rfloor \ \star\ {}^{*}(\mathsf{Tree}\ R)\ t_{rl}\ p_{rl} \end{array} \right\}$$

With our model, the tree abstraction does not need to be unfolded. The key idea is to represent this intermediate state by letting one tree keep ownership of the subtree and offer it at location q, while the other tree treats this subtree as borrowed at q. The ownership of the shared subtree can then be flexibly transferred between the two trees. Crucially, without the Offered state, as in trees with holes and cuts, such ownership transfer would not be possible since the tree currently owning the subtree would hide its location.

In the proof using our model (the full proof is given in section C), this state is represented as:

$$\left\{ \begin{array}{l} \mathsf{Tree_s}\ R\ (\mathsf{Node_s}\ x\ t_l'\ (\mathsf{Borrowed}\ (p_r :: qs_r)))\ p \\ \quad \star\ \mathsf{Tree_s}\ R\ (\mathsf{Node_s}\ x_r\ (\mathsf{pin}\ t_{rl}'\ p_{rl})\ t_{rr}')\ p_r \end{array} \right\}$$

Assume `left_rotate` returns a pointer to the root of the rotated tree. Our following specification preserves the original root pointer p. Here, rotL maps a tree to its rotated form.

$$\{\, \lfloor t = \mathsf{Node_s}\ x\ t_l'\ (\mathsf{Available}\ t_r\ qs_r) \rfloor \star \mathsf{Tree_s}\ R\ t\ p \,\}$$
$$\mathtt{left\_rotate}(p)$$
$$\{\, \lambda\ r.\ {}^{\flat}(\mathsf{Tree_s}\ R)\ (\mathsf{rotL}\ (\mathsf{Offered}\ t\ [p]))\ r \,\}$$

6 Discussion

Our logical-pinning model captures ad hoc proof patterns that occur in existing separation-logic developments: (1) the notion of holes and cut in trees (section 3.3) can be expressed uniformly as Borrowed in our model; (2) when two pointers p and q are aliases in separation logic, the common approach is to introduce a pure constraint $p = q$ and shift ownership between the aliasing pointers via equality rewriting:

$$R\ v\ p \star \lfloor p = q \rfloor \equiv R\ v\ q \star \lfloor p = q \rfloor$$

In our model, pinning captures such aliasing constraints:

$${}^{\flat}R\ (o_v, q :: qs)\ p \equiv {}^{\flat}R\ (o_v, p :: q :: qs)\ p \equiv {}^{\flat}R\ (o_v, p :: qs)\ q$$

We have demonstrated that our model enables more precise specifications. We verify the correctness of the logger example using a precise specification of `set_level_all` that guarantees the function does not relocate elements. For linked lists, we provide API specifications that precisely track all exposed pointers to elements and tails, enabling the verification of representative programs such as element-swapping.

Our model also simplifies proofs that are conceptually complex under traditional approaches. For instance, our proof of tree rotation avoids unfolding the tree representation predicate, keeping the reasoning simple: at most two trees need to be tracked, rather than two split records (each with three fields) and three subtrees.

Our model further enables proving program patterns not supported by traditional specifications, notably *expose-then-mutate*: exposing internal pointers, optionally performing container operations that do not invalidate them, and later using those pointers to read or modify container subparts. This is exactly the pattern in the previous examples (logger, element-swapping, subtree-compare).

Internal sharing within a container. Some containers allow the same subpart to be referenced multiple times within the container. For instance, the same object may be pushed to a queue more than once. In optimized trees such as compressed tries (e.g., directed acyclic word graphs) or binary decision diagrams (BDDs), distinct nodes may share a common subtree, preserving logical tree semantics while structurally forming a directed acyclic graph (DAG).

Take an example of a tree with an internally shared subtree. We can model the situation by letting one node own the subtree and offer it at a location q, while any other node that references this subtree view it as borrowed at q, analogous to the case in tree rotation (section 5.2.2). The entire structure is then viewed as a partial normal tree with one subtree absent.

Such partial containers cannot be expressed using magic wands. Magic wands do not model partial structures directly, but instead specify that the complete structure can be reconstructed once the missing part is supplied. In this case, a "complete" tree does not exist: the shared subtree cannot be simultaneously reclaimed by all referencing nodes while the heap segments owned by those nodes remain distinct.

Open questions. Our model does not answer the following questions:

- How can container-internal pointers be reasoned about in concurrent programs?
- How can container-internal pointers be reasoned about in general graphs with unrestricted sharing?
- Can our model be integrated with fractional resources [6, 7] to distinguish read-only from read–write access? A preliminary formulation of the logical borrow–return rule is:
  $({}^{\flat}R\ (\text{Offered}\ v\ (q :: qs))\ p)^{\pi}$
  $\equiv {}^{\flat}R\ (\text{Borrowed}\ (q :: qs))\ p \star ({}^{\flat}R\ (\text{Owned}\ v)\ q)^{\pi}$
  where $\pi \in (0, 1]$ is the permission fraction.
- How can container invariants be systematically tracked and enforced across borrow–return transitions?
- How can logical models, representation predicates and container-specific transition lemmas be automatically derived for containers with borrowable subparts? We suspect that existing work on contexts [3, 17, 20–22] and *lenses* [5, 13] could nicely combine with our approach.

7 Other Related Work

Beyond the work discussed in section 3, previous work differs from ours by focusing on automation [4, 11], on read-only sharing [11], or on modeling specific access patterns [4, 16].

The **iterated separating conjunction** [23, 26] generalizes $\star$ to range over sets of heap locations. It is particularly useful to specify unstructured containers, which it represents as a collection of elements with all internal pointers exposed. In contrast, logical pinning supports selective exposure.

Authoritative cameras [18, 19] in Iris maintain a coherent global view of a resource by designating an authoritative token that can mutate the state and multiple fragment tokens that provide consistent read-only views of its parts.

Rust borrows permit either one mutable reference or multiple immutable references to each object or object field, but not both. Standard mutable borrows freeze the entire container, so ad hoc APIs have been developed to allow simultaneous borrows (`pick2_mut`, `get_disjoint_mut` [27]), but they either require unsafe code, or have runtime costs, or have usage restrictions (e.g., most require obtaining all references simultaneously).

Deferred borrows [12] are a borrowing system that ensures the safety of a common workaround for the limitations of borrowing: using paths (indices in an array, keys of a map, …) instead of pointers. In combination with custom container types and custom path-returning APIs, deferred borrows enable safe path references to multiple elements in a container, with actual borrows delayed until access time, at which point the path is (re-)resolved into a pointer. In contrast, logical pinning is a purely logical discipline.

In **RefinedRust** [15], a value can be logically marked as borrowed by replacing it with a borrow name. In contrast, logical pinning records the location of the borrowed value.

8 Conclusion

Logical pinning is a new lightweight borrowing model that makes exposed internal pointers explicit in the logical representation of containers, enabling direct reasoning about pointer validity while preserving modularity. This technique unifies several ad hoc proof patterns and supports precise yet modular specifications for common idioms such as pointer caching, expose-then-mutate, and consecutive lookups.

Our case studies show that it simplifies existing proofs and allow new patterns to be verified, all without reformulating the underlying separation logic. We implemented the approach in CFML to demonstrate its soundness and practicality. Our mechanization is available online for reference.

Data Availability Statement

The formalization of the logical-pinning model and associated case studies are available at https://github.com/epfl-systemf/logical-pinning. The artifact [14] of this paper is available at https://doi.org/10.5281/zenodo.17815704.

A The Subtree-Compare Example

Let f be the logical comparison function, and allSubtreeAvl t' assert that every subtree of t' is available. The specification of `compare` uses the non-separating conjunction in its precondition to handle potentially overlapping subtrees:

$$\text{allSubtreeAvl}\ t_1' \wedge \text{allSubtreeAvl}\ t_2' \to$$
$$\{\ ({}^{\flat}(\text{Tree}_s\ R)\ t_1'\ q_1 \star H_1) \between ({}^{\flat}(\text{Tree}_s\ R)\ t_2'\ q_2 \star H_2)\ \}$$
$$\texttt{compare}(q_1, q_2)$$
$$\{\ \lambda\, r.\ \lfloor r = f\ t_1'\ t_2' \rfloor \star {}^{\flat}(\text{Tree}_s\ R)\ t_2'\ q_2 \star H_2\ \}$$

Starting from $^\flat(\mathrm{Tree_s}\ R)\ t'\ p$, the state after performing the two `lookup` operations is as follows:

$$^\flat(\mathrm{Tree_s}\ R)\ t'[\![k_1 := \mathsf{pin}\ t'[k_1]\ q_1]\!][\![k_2 := \mathsf{pin}\ t'[k_2]\ q_2]\!]\ p$$

In unification with `compare`'s precondition, H_1 is the partial tree with $t'[k_1]$ borrowed (H_2 is obtained analogously):

$$^\flat(\mathrm{Tree_s}\ R)\ t'[\![k_1 := \mathsf{Borrowed}\ [q_1]]\!]\ p$$

A logical return step after `compare` can restore t'.

B Proof of Swapping Two Elements in a List

Assume `memswap` swaps the contents of the cells pointed to by two distinct pointers, or does nothing if given the same pointer. In the following, we give the correctness proof of the element-swapping program; due to space constraints, we omit the case where $n_1 = n_2$, which follows the same proof style. In the proof, we highlight logical borrow steps, logical return steps and logical forget steps using different colors.

$n_1 \neq n_2 \land n_1 < |l| \land n_2 < |l| \to$
$\mathrm{nth}\ n_1\ l = \mathsf{Available}\ v_1\ qs_1 \land \mathrm{nth}\ n_2\ l = \mathsf{Available}\ v_2\ qs_2 \to$

$\{\ \mathrm{CList_e}\ l\ p\ \}$
`let` $q_1 \triangleq$ `nth_elem_ptr(p, n1) in`
$\{\ \mathrm{CList_e}\ l[n_1 := \mathsf{Offered}\ v_1\ (q_1 :: qs_1)]\ p\ \}$
`let` $q_2 \triangleq$ `nth_elem_ptr(p, n2) in`
$\left\{ \begin{array}{l} \mathrm{CList_e}\ l[n_1 := \mathsf{Offered}\ v_1\ (q_1 :: qs_1)] \\ \qquad [n_2 := \mathsf{Offered}\ v_2\ (q_2 :: qs_2)]\ p \end{array} \right\}$
$\equiv \left\{ \begin{array}{l} \mathrm{CList_e}\ l[n_1 := \mathsf{Borrowed}\ (q_1 :: qs_1)] \\ \qquad [n_2 := \mathsf{Borrowed}\ (q_2 :: qs_2)]\ p \\ \quad \star\ \mathsf{Cell}\ v_1\ q_1 \star \mathsf{Cell}\ v_2\ q_2 \end{array} \right\}$
`memswap(`q_1`, `q_2`)`
$\left\{ \begin{array}{l} \mathrm{CList_e}\ l[n_1 := \mathsf{Borrowed}\ (q_1 :: qs_1)] \\ \qquad [n_2 := \mathsf{Borrowed}\ (q_2 :: qs_2)]\ p \\ \quad \star\ \mathsf{Cell}\ v_2\ q_1 \star \mathsf{Cell}\ v_1\ q_2 \end{array} \right\}$
$\equiv \left\{ \begin{array}{l} \mathrm{CList_e}\ l[n_1 := \mathsf{Offered}\ v_2\ (q_1 :: qs_1)] \\ \qquad [n_2 := \mathsf{Offered}\ v_1\ (q_2 :: qs_2)]\ p \end{array} \right\}$

C Proof of Binary-Tree Left Rotation

$\{\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Available}\ t_r\ qs_r))\ p\ \}$
`let pr := p.rtree in`
$\{\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Offered}\ t_r\ (p_r :: qs_r)))\ p\ \}$
$\equiv \left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ (p_r :: qs_r)))\ p \\ \quad \star\ \mathrm{Tree_s}\ R\ t_r\ p_r \end{array} \right\}$
`if not (_is_empty pr) then (`
$\left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ (p_r :: qs_r)))\ p \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ t'_{rl}\ t'_{rr})\ p_r \end{array} \right\}$
`let prl := pr.ltree in`
$\left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ (p_r :: qs_r)))\ p \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ (\mathsf{pin}\ t'_{rl}\ p_{rl})\ t'_{rr})\ p_r \end{array} \right\}$
`p.rtree <- prl`
$\left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ [p_{rl}]))\ p \\ \quad \star \exists q.\ ^\flat(\mathrm{Tree_s}\ R)\ (\mathsf{Borrowed}\ (p_r :: qs_r))\ q \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ (\mathsf{pin}\ t'_{rl}\ p_{rl})\ t'_{rr})\ p_r \end{array} \right\}$
$\equiv \left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ [p_{rl}]))\ p \\ \quad \star \lfloor \forall\ q \in qs_r.\ q = p_r \rfloor \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ (\mathsf{pin}\ t'_{rl}\ p_{rl})\ t'_{rr})\ p_r \end{array} \right\}$
`pr.ltree <- p`
$\left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ [p_{rl}]))\ p \\ \quad \star \lfloor \forall\ q \in qs_r.\ q = p_r \rfloor \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ (\mathsf{Borrowed}\ [p])\ t'_{rr})\ p_r \\ \quad \star \exists q.\ ^\flat(\mathrm{Tree_s}\ R)\ (\mathsf{pin}\ t'_{rl}\ p_{rl})\ q \end{array} \right\}$
$\vdash \left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ [p_{rl}]))\ p \\ \quad \star \lfloor \forall\ q \in qs_r.\ q = p_r \rfloor \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ (\mathsf{Borrowed}\ [p])\ t'_{rr})\ p_r \\ \quad \star\ ^\flat(\mathrm{Tree_s}\ R)\ t'_{rl}\ p_{rl} \end{array} \right\}$
$\equiv \left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{pin}\ t'_{rl}\ p_{rl}))\ p \\ \quad \star \lfloor \forall\ q \in qs_r.\ q = p_r \rfloor \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ (\mathsf{Borrowed}\ [p])\ t'_{rr})\ p_r \end{array} \right\}$
$\vdash \left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ t'_{rl})\ p \\ \quad \star \lfloor \forall\ q \in qs_r.\ q = p_r \rfloor \\ \quad \star\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r\ (\mathsf{Borrowed}\ [p])\ t'_{rr})\ p_r \end{array} \right\}$
$\equiv \left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x_r \\ \qquad (\mathsf{Offered}\ (\mathrm{Node_s}\ x\ t'_l\ t'_{rl})\ [p])\ t'_{rr})\ p_r \\ \quad \star \lfloor \forall\ q \in qs_r.\ q = p_r \rfloor \end{array} \right\}$
$\equiv \{\ ^\flat(\mathrm{Tree_s}\ R)\ (\mathrm{rotL}\ (\mathsf{Offered}\ t\ (p :: [])))\ p_r\ \}$
`) else p`
$\left\{ \begin{array}{l} \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Borrowed}\ (p_r :: qs_r)))\ p \\ \quad \star\ \mathrm{Tree_s}\ R\ \mathrm{Leaf_s}\ p_r \end{array} \right\}$
$\equiv \{\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Offered}\ \mathrm{Leaf_s}\ (p_r :: qs_r)))\ p\ \}$
$\vdash \{\ \mathrm{Tree_s}\ R\ (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Available}\ \mathrm{Leaf_s}\ qs_r))\ p\ \}$
$\equiv \left\{ \begin{array}{l} ^\flat(\mathrm{Tree_s}\ R)\ (\mathsf{Offered} \\ \qquad (\mathrm{Node_s}\ x\ t'_l\ (\mathsf{Available}\ \mathrm{Leaf_s}\ qs_r))\ [p])\ p \end{array} \right\}$
$\equiv \{\ ^\flat(\mathrm{Tree_s}\ R)\ (\mathrm{rotL}\ (\mathsf{Offered}\ t\ [p]))\ p_r\ \}$

Acknowledgments

We thank Jade Philipoom for early discussions on this topic. We also thank Aurèle Barrière, Arthur Charguéraud, Gregory Malecha, Alexandre Moine, François Pottier, and the anonymous reviewers for their valuable feedback and suggestions.

References

[1] 2025. Containers Library - Cppreference.Com. Retrieved 2025-07-23 from https://en.cppreference.com/w/cpp/container.html#Iterator_invalidation

[2] 2025. Spdlog: Fast C++ Logging Library. Retrieved 2025-07-23 from https://github.com/gabime/spdlog/tree/v1.x

[3] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2003. Derivatives of Containers. In *Typed Lambda Calculi and Applications*, Martin Hofmann (Ed.). Springer, Berlin, Heidelberg, 16–30. doi:10.1007/3-540-44904-3_2

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vol. 3. ACM, 147:1–147:30. doi:10.1145/3360573

[5] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 407–419. doi:10.1145/1328438.1328487

[6] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. Association for Computing Machinery, New York, NY, USA, 259–270. doi:10.1145/1040305.1040327

[7] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 55–72. doi:10.5555/1760267.1760273

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

[9] Arthur Charguéraud. 2016. Higher-Order Representation Predicates in Separation Logic. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/2854065.2854068

[10] Arthur Charguéraud. 2023. *A Modern Eye on Separation Logic for Sequential Programs*. Thesis. Université de Strasbourg. https://inria.hal.science/tel-04076725

[11] Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. 2020. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 141–168. doi:10.1007/978-3-030-44914-8_6

[12] Chris Fallin. 2020. Safe, Flexible Aliasing with Deferred Borrows. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:26. doi:10.4230/LIPIcs.ECOOP.2020.30

[13] John Nathan Foster. 2009. *Bidirectional Programming Languages*. Ph. D. Dissertation. University of Pennsylvania.

[14] Yawen Guan and Clément Pit-Claudel. 2025. *Artifact: Precise Reasoning about Container-Internal Pointers with Logical Pinning*. doi:10.5281/zenodo.17815704

[15] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 192:1115–192:1139. doi:10.1145/3656422

[16] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 116:711–116:741. doi:10.1145/3547647

[17] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (Sept. 1997), 549–554. doi:10.1017/S0956796897002864

[18] 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 (Jan. 2018), e20. doi:10.1017/S0956796818000151

[19] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. doi:10.1145/2676726.2676980

[20] Daan Leijen and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 40:1152–40:1181. doi:10.1145/3571233

[21] Conor McBride. 2001. The Derivative of a Regular Type Is Its Type of One-Hole Contexts. (2001). http://strictlypositive.org/diff.pdf

[22] Yasuhiko Minamide. 1998. A Functional Representation of Data Structures with a Hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. Association for Computing Machinery, New York, NY, USA, 75–84. doi:10.1145/268946.268953

[23] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 405–425. doi:10.1007/978-3-319-41528-4_22

[24] Peter O'Hearn. 2019. Separation Logic. *Commun. ACM* 62, 2 (Jan. 2019), 86–95. doi:10.1145/3211968

[25] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, Berlin, Heidelberg, 1–19. doi:10.5555/647851.737404

[26] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

[27] The Rust Project Developers. 2025. Slice::Get_disjoint_mut - The Rust Standard Library. Retrieved 2025-09-12 from https://doc.rust-lang.org/std/primitive.slice.html#method.get_disjoint_mut

[28] The Rust Project Developers. 2025. Splitting Borrows - The Rustonomicon. Retrieved 2025-09-12 from https://doc.rust-lang.org/nomicon/borrow-splitting.html

[29] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying Graph-Manipulating C Programs via Localizations within Data Structures. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 171:1–171:30. doi:10.1145/3360597

Received 2025-09-03; accepted 2025-11-13