The Lattice Phase System: First-Class Immutability with Dual-Heap Memory Management

Alex Jokela
alex.c.jokela@gmail.com

February 2026

Abstract

Dynamic languages typically treat mutability as the default and offer immutability only as an advisory annotation (JavaScript's `Object.freeze`) or through separate type hierarchies (Scala's `val`/`var`). We present *Lattice*, a dynamically typed language whose runtime associates every value with a *phase tag*—either fluid (mutable) or crystal (deeply immutable)—and mediates all transitions through explicit **freeze**, **thaw**, **clone**, and **forge** operations. The phase system is backed by a *dual-heap memory architecture*: a garbage-collected fluid heap for mutable data and an arena-based region store for crystal (frozen) data, connected by a freeze migration protocol that deep-clones values into arena regions and establishes full pointer disjointness between heaps.

We formalize the phase system with a big-step operational semantics and static phase-checking rules, and prove six key safety properties: phase monotonicity, value isolation, consuming freeze semantics, forge soundness, heap separation, and thaw independence. Empirical evaluation across eight benchmarks shows that the dual-heap architecture adds less than 2% wall-clock overhead compared to a single-heap baseline, with RSS costs proportional to frozen data volume. The architecture provides a foundation for future optimizations including copy-on-write sharing, concurrent collection, and safe parallelism via the crystal-implies-shareable invariant.

1 Introduction

Mutable state is a perennial source of bugs in dynamic languages. Aliasing permits distant mutations to corrupt shared data structures; defensive copying wastes memory and CPU cycles; and garbage collectors must trace the entire heap regardless of whether values are long-lived and unchanging. Consider a game engine that periodically snapshots its state for rollback:

In a conventional dynamic language, the programmer must manually ensure that checkpoints are truly independent of the live state—a single shared reference suffices to corrupt the snapshot. JavaScript's `Object.freeze` is shallow and advisory; Python has no built-in mechanism; and even Clojure's persistent data structures require the programmer to consistently use the functional API.

Lattice addresses this problem by making immutability a *first-class runtime property*. Every value carries a phase tag—fluid for mutable, crystal for deeply immutable—and the runtime enforces phase discipline through explicit transition operations. In Figure 1, `freeze(clone(state))` produces a crystal checkpoint: a deeply immutable, independently allocated snapshot that the garbage collector can skip during mark traversal and that can be bulk-deallocated when the checkpoint becomes unreachable.

```
struct GameState {
    frame: Int, score: Int,
    player_x: Int, player_y: Int,
    hp: Int, entities: Array,
}

fn main() {
    let checkpoints = []
    let state = GameState {
        frame: 0, score: 0,
        player_x: 100, player_y: 100,
        hp: 100, entities: make_entities(20),
    }
    for f in 0..500 {
        state = tick(state)
        // Checkpoint every 25 frames
        if state.frame % 25 == 0 {
            let cp = freeze(clone(state))
            checkpoints.push(cp)
        }
        // Rollback on desync
        if state.frame % 100 == 0 {
            if checkpoints.len() > 1 {
                let idx = checkpoints.len() - 2
                state = thaw(checkpoints[idx])
            }
        }
    }
}
```

Listing 1: Game rollback in Lattice: mutable state is checkpointed by freezing and restored by thawing.

Contributions. This paper makes three contributions:

1. **The Lattice phase system** (§2–§3.3): a runtime discipline where every value is tagged fluid or crystal, with explicit **freeze**/**thaw**/**clone**/**forge** operations mediating all phase transitions. A static phase checker provides early error detection in strict mode.
2. **A dual-heap memory architecture** (§5): a garbage-collected fluid heap paired with an arena-based region store, connected by a freeze migration protocol that establishes full pointer disjointness. We prove six safety properties (§4), including heap separation (no crystal pointer appears in the fluid allocation list) and phase monotonicity (crystal values cannot be mutated in place). Full proofs appear in Appendix A.
3. **Empirical validation** (§6): across eight benchmarks spanning allocation-heavy, closure-heavy, event-sourcing, and snapshot-based workloads, the dual-heap architecture adds less than 2% wall-clock overhead to the two heaviest benchmarks (event sourcing at 1.0 s and game rollback at 1.9 s), with RSS costs proportional to frozen data volume.

2 Language Overview

Lattice is a dynamically typed, expression-oriented language with C-like syntax, first-class closures, structs, arrays, and maps. Its distinguishing feature is the *phase system*: every runtime value carries a phase tag that is either fluid (mutable) or crystal (deeply immutable).

2.1 Declarations and Phase Tags

Lattice provides three declaration keywords that determine the initial phase of a binding:

- `flux x = e` — declares x with phase fluid (mutable). The value may be reassigned and its sub-values may be mutated in place.
- `fix x = e` — declares x with phase crystal (immutable). The initializer is automatically frozen and migrated to the region store. Any attempt to reassign or mutate x is rejected.
- `let x = e` — infers the phase from the initializer (casual mode) or is rejected in strict mode, where the programmer must choose explicitly.

A simple example:

```
#mode strict
flux counter = 0        // fluid: mutable
counter += 1            // OK
fix pi = freeze(3.14)   // crystal: immutable
// pi = 2.71            // ERROR: cannot assign to crystal
flux thawed = thaw(pi) // deep clone into fluid heap
thawed = 2.71           // OK: thawed is fluid
```

Listing 2: Phase declarations and transitions.

2.2 Phase Transition Operations

Four operations mediate phase transitions:

`freeze(e)` Evaluates e, sets all phase tags to crystal recursively, deep-clones the value into an arena region, and frees the original fluid pointers. In strict mode, freezing a variable *consumes* the binding (use-after-freeze is an error).

`thaw(e)` Deep-clones the value and sets all phase tags to fluid. The original crystal value is unaffected.

`clone(e)` Deep-clones the value, preserving the original phase. Useful for creating independent copies of mutable data.

`forge { ... }` A block expression that evaluates its body, then automatically freezes the result. Forge blocks are *compositional factories*: their output is guaranteed crystal.

Figure 1 shows the state diagram. The key design principle is that *all transitions are explicit and deep*: freezing recursively freezes all sub-values, and thawing produces a fully independent mutable copy.

2.3 Strict Mode and Static Checking

Lattice programs operate in one of two modes:

- **Casual mode** (default): `let` is permitted, phase violations produce warnings, and freeze updates bindings in place.
- **Strict mode** (`#mode strict`): `let` is rejected (use `flux` or `fix`), crystal assignments are errors, and freeze *consumes* the binding.

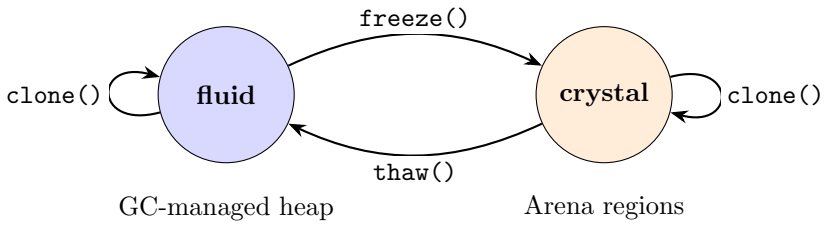

Figure 1: Phase transition state diagram. `freeze` migrates values from the fluid heap to arena regions; `thaw` deep-clones back. `clone` produces independent copies within the same heap.

Strict mode provides a lightweight form of affine resource management: **freeze** : $\tau^{\mathsf{fluid}} \multimap \tau^{\mathsf{crystal}}$, where the linear arrow $\multimap$ indicates that the fluid binding is consumed. This prevents use-after-freeze bugs without requiring a full linear type system.

2.4 Event Sourcing Example

Figure 3 demonstrates an event-sourcing pattern. Each event is frozen immediately upon creation, forming an append-only log of crystal values. State is rebuilt by thawing and replaying events. The phase system guarantees that events are never accidentally mutated after insertion into the log, and the dual-heap architecture ensures that long-lived frozen events reside in arena regions that are bulk-deallocated when the log is discarded.

2.5 Forge Blocks

Forge blocks provide a controlled-mutation pattern for constructing immutable data:

```
fix config = forge {
    flux temp = Map::new()
    temp.set("host", "localhost")
    temp.set("port", "8080")
    temp.set("debug", "true")
    freeze(temp)
}
// config is crystal --- mutations are rejected
```

Listing 4: Forge block for constructing immutable configuration.

The body of a forge block may use arbitrary mutable operations; the result is automatically frozen. Forge blocks compose: nested forge blocks each produce crystal output, and the outer forge's freeze on an already-crystal value is idempotent.

3 Formal Semantics

We present the formal semantics of the Lattice phase system, comprising semantic domains, abstract syntax, static phase-checking rules, big-step operational semantics, and the dual-heap memory model.

3.1 Phase Tags and Semantic Domains

Definition 3.1 (Phase Tags)**.** *The set of* phase tags *is:* $\sigma \in \mathsf{Phase} \triangleq \{\mathsf{fluid}, \mathsf{crystal}, \bot\}$*, where* fluid *denotes mutable data,* crystal *denotes deeply immutable data, and* $\bot$ *(*$\bot$*) denotes data whose phase has not been explicitly specified.*

Definition 3.2 (Tagged Values)**.** *A* tagged value *is a pair* v^σ *where* v *is a raw value and* $\sigma \in \mathsf{Phase}$ *is its phase tag. The raw value domain is:*

$$v \in \mathsf{Val} ::= n \mid r \mid b \mid s \mid [v_1, \ldots, v_k] \mid S\{f_1 : v_1, \ldots, f_k : v_k\} \mid \langle \bar{x}, e, \rho \rangle \mid () \mid \{s_1 : v_1, \ldots, s_k : v_k\}$$

where $n \in \mathbb{Z}$*,* $r \in \mathbb{R}$*,* $b \in \mathbb{B}$*,* $s \in \mathsf{String}$*, and* $\langle \bar{x}, e, \rho \rangle$ *is a closure with parameters* $\bar{x}$*, body* e*, and captured environment* ρ*.*

Definition 3.3 (Environments and Stores)**.** *An* environment $\rho : \mathsf{Var} \rightharpoonup \mathsf{TVal}$ *maps variable names to tagged values. Variable lookup produces a* deep clone*:* $\rho(x) = \mathsf{deepclone}(\rho_{\text{raw}}(x))$*. This ensures* value isolation*: the caller receives an independent copy.*

Definition 3.4 (Dual-Heap Store)**.** *The* store $\mu = (\mathcal{F}, \mathcal{R})$ *is a disjoint union of a* fluid heap $\mathcal{F} \in \mathsf{FHeap}$ *(tracked allocations managed by mark-sweep GC; all* fluid*- and* $\bot$*-tagged values reside here) and a* region store $\mathcal{R} \in \mathsf{RStore} = \mathsf{RId} \rightharpoonup \mathsf{Region}$ *(arena-allocated regions; all* crystal*-tagged values reside in exactly one region).*

3.2 Abstract Syntax

$d \in \mathsf{Decl}$	::=	$\mathbf{flux}\ x = e \mid \mathbf{fix}\ x = e \mid \mathbf{let}\ x = e$	(declarations)
$e \in \mathsf{Expr}$	::=	$x \mid c \mid e_1 \oplus e_2 \mid [e_1, \ldots, e_k] \mid S\{f_1 : e_1, \ldots\}$	(base expressions)
	$\mid$	$\lvert x_1, \ldots, x_k \rvert\ \{\ \bar{s}\ \}$	(closures)
	$\mid$	$e.f \mid e[e'] \mid e(e_1, \ldots, e_k)$	(access, index, call)
	$\mid$	$\mathbf{freeze}(e) \mid \mathbf{thaw}(e) \mid \mathbf{clone}(e)$	(phase operations)
	$\mid$	$\mathbf{forge}\ \{\ \bar{s}\ \}$	(forge block)
$s \in \mathsf{Stmt}$	::=	$d \mid x = e \mid e \mid \mathbf{return}\ e$	(statements)
	$\mid$	$\mathbf{for}\ x\ \mathbf{in}\ e\ \{\ \bar{s}\ \} \mid \mathbf{while}\ e\ \{\ \bar{s}\ \}$	(loops)

Programs operate in mode $\mathcal{M} \in \{\mathsf{casual}, \mathsf{strict}\}$.

3.3 Static Phase Checking

The static phase checker maintains a context $\Gamma : \mathsf{Var} \rightharpoonup \mathsf{Phase}$ and infers phases for expressions via $\Gamma \vdash e : \sigma$ ("under Γ, expression e has phase σ").

Figure 2 shows the key rules. The PC-Spawn-Strict rule is particularly noteworthy: it prevents fluid (mutable) bindings from being captured across thread boundaries, enforcing that shared data must be crystal. This provides a foundation for safe parallelism.

3.4 Operational Semantics

We define a big-step semantics with store-passing. Configurations have the form $\langle \rho, \mu, e \rangle$ and evaluate to $\langle \mu', v^\sigma \rangle$.

3.4.1 Variables and Literals

$$\text{E-Lit}\quad \frac{c \text{ is a literal}}{\langle \rho, \mu, c\rangle \Downarrow \langle \mu, c^{\perp}\rangle} \qquad \text{E-Var}\quad \frac{\rho_{\text{raw}}(x) = v^{\sigma} \qquad v'^{\sigma} = \mathsf{deepclone}(v^{\sigma})}{\langle \rho, \mu, x\rangle \Downarrow \langle \mu, v'^{\sigma}\rangle}$$

E-Var deep-clones the stored value, ensuring value isolation: every variable read yields an independent copy.

3.4.2 Phase Transitions

Figure 3 contrasts strict and casual freeze. The critical difference is that strict-mode freeze removes the binding $(\rho \setminus x)$, preventing use-after-freeze.

Thaw deep-clones a value and sets its phase to fluid:

$$\text{E-Thaw-Var}\quad \frac{\rho_{\text{raw}}(x) = v^{\sigma} \qquad v'^{\mathsf{fluid}} = \mathsf{deepclone}(v^{\sigma})[\sigma := \mathsf{fluid}] \qquad \rho' = \rho[x \mapsto v'^{\mathsf{fluid}}]}{\langle \rho, \mu, \mathbf{thaw}(x)\rangle \Downarrow \langle \rho', \mu, \mathsf{deepclone}(v'^{\mathsf{fluid}})\rangle}$$

Clone preserves the original phase:

$$\text{E-Clone}\quad \frac{\langle \rho, \mu, e\rangle \Downarrow \langle \mu', v^{\sigma}\rangle \qquad v'^{\sigma} = \mathsf{deepclone}(v^{\sigma})}{\langle \rho, \mu, \mathbf{clone}(e)\rangle \Downarrow \langle \mu', v'^{\sigma}\rangle}$$

Forge evaluates its body and freezes the result:

$$\text{E-Forge}\quad \frac{\langle \rho', \mu, \bar{s}\rangle \Downarrow_{block} \langle \mu', v^{\sigma}\rangle \qquad v'^{\mathsf{crystal}} = \mathsf{setphase}(v^{\sigma}, \mathsf{crystal}) \qquad (v''^{\mathsf{crystal}}, \mathcal{R}') = \mathsf{freeze\_region}(v'^{\mathsf{crystal}}, \mathcal{R}')}{\langle \rho, \mu, \mathbf{forge}\ \{\ \bar{s}\ \}\rangle \Downarrow \langle (\mathcal{F}', \mathcal{R}'), v''^{\mathsf{crystal}}\rangle}$$

3.4.3 Binding Declarations

$$\text{E-Flux}\quad \frac{\langle \rho, \mu, e\rangle \Downarrow \langle \mu', v^{\sigma}\rangle \qquad \mathcal{M} = \mathsf{strict} \Rightarrow \sigma \neq \mathsf{crystal}}{\langle \rho, \mu, \mathbf{flux}\ x = e\rangle \Downarrow \langle \rho[x \mapsto v^{\mathsf{fluid}}], \mu', ()\rangle}$$

$$\text{E-Fix}\quad \frac{\langle \rho, \mu, e\rangle \Downarrow \langle \mu', v^{\sigma}\rangle \qquad v'^{\mathsf{crystal}} = \mathsf{setphase}(v^{\sigma}, \mathsf{crystal}) \qquad (v''^{\mathsf{crystal}}, \mathcal{R}') = \mathsf{freeze\_region}(v'^{\mathsf{crystal}}, \mathcal{R}')}{\langle \rho, \mu, \mathbf{fix}\ x = e\rangle \Downarrow \langle \rho[x \mapsto v''^{\mathsf{crystal}}], (\mathcal{F}', \mathcal{R}'), ()\rangle}$$

3.4.4 Assignment and Lvalue Resolution

Assignment uses *lvalue resolution* to obtain a mutable pointer into the environment:

$$\text{E-Assign}\quad \frac{\langle \rho, \mu, e\rangle \Downarrow \langle \mu', v^{\sigma}\rangle \qquad \rho_{\text{raw}}(x) = v_0^{\sigma_0} \qquad \mathcal{M} = \mathsf{strict} \Rightarrow \sigma_0 \neq \mathsf{crystal}}{\langle \rho, \mu, x = e\rangle \Downarrow \langle \rho[x \mapsto v^{\sigma}], \mu', ()\rangle}$$

$$\text{E-Assign-Crystal-Err}\quad \frac{\mathcal{M} = \mathsf{strict} \qquad \rho_{\text{raw}}(x) = v_0^{\mathsf{crystal}}}{\langle \rho, \mu, x = e\rangle \Downarrow \mathsf{error}(\text{“cannot assign to crystal binding”})}$$

The read/write asymmetry is a defining characteristic: *reads* always deep-clone (via E-VAR), ensuring value isolation, while *writes* resolve lvalue paths to direct pointers, enabling efficient in-place mutation of fluid data.

3.5 Memory Model

3.5.1 Fluid Heap

The fluid heap $\mathcal{F} = \{(p_i, n_i, m_i)\}_{i \in I}$ is a set of tracked allocations where p_i is a pointer, n_i is the allocation size, and $m_i \in \{0, 1\}$ is the GC mark bit. The garbage collector uses a three-phase mark-sweep protocol: (1) *unmark* all allocations; (2) *mark* all reachable values from root environments, saved caller environments, and the shadow stack—for crystal values with a valid region ID, record the region ID and return immediately; (3) *sweep* unmarked fluid allocations and collect unreachable regions.

3.5.2 Region Store and Arena Allocation

Definition 3.5 (Crystal Region)**.** *A crystal region $R = (r, \varepsilon, P, n)$ consists of a unique identifier $r \in \mathsf{RId}$, an epoch ε, a linked list of arena pages P (each 4096 bytes), and total bytes used n. Arena allocation is $O(1)$ amortized (bump-pointer within a page).*

Definition 3.6 (Region Collection)**.** *Given reachable region IDs $\mathcal{S} \subseteq \mathsf{RId}$ from the mark phase: $\mathsf{region\_collect}(\mathcal{R}, \mathcal{S}) = \{r \mapsto R \mid (r \mapsto R) \in \mathcal{R},\ r \in \mathcal{S}\}$. Unreachable regions are freed in $O(|P|)$ time per region—bulk deallocation without per-object overhead.*

3.5.3 Freeze Migration Protocol

The central operation linking the two heaps is $\mathsf{freeze\_region}(v^{\mathsf{crystal}}, (\mathcal{F}, \mathcal{R}))$:

1. Create a fresh region R with identifier r.
2. Set the global arena pointer to R.
3. Deep-clone v into R: all allocations route through $\mathsf{arena\_alloc}(R, \cdot)$.
4. Reset the arena pointer to null.
5. Set $\mathsf{region\_id} := r$ on v' and all sub-values.
6. Free the original fluid-heap pointers.
7. Return v'^{crystal} with updated $\mathcal{R}$.

After migration, the crystal value has completely independent pointers from $\mathcal{F}$.

4 Properties and Proof Sketches

We state six key safety properties. Full proofs appear in Appendix A.

Table 1 summarizes the six theorems. We now state each formally and provide proof sketches.

Theorem 4.1 (Phase Monotonicity)**.** *If v^{crystal} is a crystal-tagged value bound to variable x, then for any assignment targeting x or a sub-path of x, evaluation in strict mode produces* error*.*

Proof sketch. By exhaustive case analysis on all mutation paths: (I) direct assignment is blocked by E-ASSIGN-CRYSTAL-ERR; (II) compound lvalue assignment is blocked by the post-resolution crystal guard, which holds by induction on path length using the recursive phase invariant of setphase; (III) mutating methods check crystal phase on the receiver; (IV) the static rule PC-ASSIGN rejects crystal assignments at analysis time. See Appendix A.1. □

Theorem	Statement (informal)
Phase Monotonicity	Crystal values cannot be mutated in place: all assignment paths and mutating methods are blocked by runtime guards (strict mode) and static analysis.
Value Isolation	Variable reads produce independent copies: the returned value shares no mutable state with the environment's stored copy.
Consuming Freeze	In strict mode, freezing a variable removes it from the environment, preventing use-after-freeze.
Forge Soundness	The result of a forge block is always crystal-phased, regardless of the body's internal mutations.
Heap Separation	No pointer reachable from a crystal value with a valid region ID appears in the fluid heap's allocation list.
Thaw Independence	Thawing produces a fresh copy that shares no state with the source; the original crystal region is unaffected.

Table 1: The six safety properties of the Lattice phase system.

Theorem 4.2 (Value Isolation)**.** *If* $\langle \rho, \mu, x \rangle \Downarrow \langle \mu, v'^{\sigma} \rangle$*, then* v' *shares no mutable state with* $\rho_{\text{raw}}(x)$*:* $\mathsf{mreach}(v'^{\sigma}) \cap \mathsf{mreach}(v^{\sigma}) = \emptyset$.

Proof sketch. By the E-Var rule, $v' = \mathsf{deepclone}(v)$. The Deep Clone Independence Lemma shows that $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$ by structural induction on value types. Any shared arena addresses are excluded from mutable reachability by Phase Monotonicity. See Appendix A.2. □

Theorem 4.3 (Consuming Freeze)**.** *In strict mode, after evaluating* **freeze**(x)*:* $x \notin \mathrm{dom}(\rho')$.

Proof sketch. Direct from the E-Freeze-Strict rule: the resulting environment is $\rho' = \rho \setminus x$, and by definition of environment removal, $x \notin \mathrm{dom}(\rho')$. Subsequent access to x fails because E-Var requires $x \in \mathrm{dom}(\rho')$. See Appendix A.3. □

Theorem 4.4 (Forge Soundness)**.** $\langle \rho, \mu, \textbf{forge}\ \{\ \bar{s}\ \} \rangle \Downarrow \langle \mu', v^{\sigma} \rangle \Longrightarrow \sigma = \mathsf{crystal}$.

Proof sketch. By E-Forge, the body's result undergoes setphase (which is total and forces crystal by the Setphase Totality Lemma) followed by freeze_region (which preserves crystal). Error and signal cases hold vacuously. See Appendix A.4. □

Theorem 4.5 (Heap Separation)**.** $\forall\, v^{\mathsf{crystal}}$ *with region* r*:* $\forall\, a \in \mathsf{ptrs}(v)$*:* $a \notin \mathrm{dom}(\mathcal{F})$.

Proof sketch. By induction on the seven steps of the freeze migration protocol. Arena pages are allocated via direct `malloc`, bypassing fluid_alloc; the Arena Routing Completeness Lemma shows all allocations during clone go to the arena; the Arena–Fluid Disjointness Lemma shows no arena address appears in $\mathcal{F}$. The original fluid pointers are freed in step 6. See Appendix A.5. □

Theorem 4.6 (Thaw Independence)**.** $\langle \rho, \mu, \textbf{thaw}(e) \rangle \Downarrow \langle \mu', v'^{\mathsf{fluid}} \rangle \Longrightarrow \mathsf{ptrs}(v') \cap \mathsf{ptrs}([\![e]\!]) = \emptyset$.

Proof sketch. Since no freeze is in progress, $\mathsf{g\_arena} = \mathsf{null}$, so all allocations in the deep clone go to the fluid heap. Fresh pointers are guaranteed disjoint from the source by the allocator contract. setphase modifies only phase tags, not pointers. See Appendix A.6. □

5 Implementation

Lattice is implemented in approximately 8,000 lines of C, comprising a hand-written recursive-descent parser, a tree-walking evaluator, and the dual-heap memory subsystem. We describe the key implementation decisions.

5.1 Dual-Heap Architecture

Figure 4 shows the architecture. The `DualHeap` structure contains a `FluidHeap` (linked list of tracked allocations with mark bits) and a `RegionManager` (dynamic array of `CrystalRegion` pointers).

5.2 Global Arena Routing

The key implementation technique is a *global arena pointer* (`g_arena`). All value allocation functions—`lat_alloc`, `lat_calloc`, `lat_strdup`—check this pointer first:

```
static CrystalRegion *g_arena = NULL;

static void *lat_alloc(size_t size) {
    if (g_arena) return arena_alloc(g_arena, size);
    if (g_heap)  return fluid_alloc(g_heap->fluid, size);
    return malloc(size);
}
```

Listing 5: Arena routing in `value.c`.

During `freeze_to_region`, the arena pointer is set before deep-cloning and reset after, ensuring that *every* allocation in the clone—including strings, arrays, struct fields, map entries, and closure environments—goes into the arena. This is what establishes the heap separation invariant (Theorem 4.5).

5.3 Garbage Collection with Crystal Fast-Path

The GC mark phase traverses all reachable values. When it encounters a crystal value with a valid region ID, it records the region ID in a reachable set and *returns immediately*, without following any pointers:

```
static void gc_mark_value(FluidHeap *fh, LatValue *v,
                          LatVec *reachable_regions) {
    if (v->phase == VTAG_CRYSTAL
        && v->region_id != (size_t)-1) {
        lat_vec_push(reachable_regions, &v->region_id);
        return;  // skip: all ptrs are in arena
    }
    // ... mark fluid pointers recursively
}
```

Listing 6: Crystal fast-path in `gc_mark_value`.

This fast-path is safe because heap separation guarantees no crystal pointer appears in the fluid allocation list. The fast-path also reduces GC pause time proportionally to the fraction of the object graph that is crystal.

5.4 Freeze Walkthrough

The complete `freeze_to_region` function is 15 lines of C:

```
static void freeze_to_region(Evaluator *ev, LatValue *v) {
    if (ev->no_regions) return;
    CrystalRegion *region = region_create(
        ev->heap->regions);       // step 1
    value_set_arena(region);      // step 2
    LatValue clone = value_deep_clone(v); // step 3
    value_set_arena(NULL);        // step 4
    set_region_id_recursive(&clone, region->id); // step 5
    value_free(v);                // step 6
    *v = clone;                   // step 7
}
```

Listing 7: The freeze migration protocol in `eval.c`.

The `no_regions` flag enables a single-heap mode (`--no-regions`) used as the baseline in our evaluation. In debug builds, `assert_crystal_not_fluid` validates the heap separation invariant after every GC cycle.

6 Evaluation

We evaluate the dual-heap architecture empirically, comparing the full region-based implementation against a single-heap baseline (`--no-regions`) that retains the phase system but skips arena migration.

6.1 Experimental Setup

Platform. All experiments run on macOS (Darwin 25.2.0) with Apple Silicon. The Lattice interpreter is compiled with `clang -O2`.

Methodology. Each benchmark runs 20 times per mode (regions vs. no-regions). We report mean wall-clock time with standard deviation, and peak RSS from the interpreter's internal memory statistics. The `--no-regions` flag disables arena allocation: `freeze` sets the phase tag but does not create a region or deep-clone into arena pages. This isolates the cost of the dual-heap mechanism itself.

6.2 Benchmark Suite

Table 2 describes the eight benchmarks. They span four categories: (1) allocation-heavy workloads without phase operations (alloc_churn, closure_heavy), serving as controls to verify that the dual-heap infrastructure imposes no overhead when unused; (2) event-sourcing patterns with many freezes and replays; (3) snapshot-based workloads (game rollback, undo/redo, persistent tree); and (4) a micro-benchmark stressing the freeze/thaw hot path.

6.3 Wall-Clock Results

Figure 6 and Table 3 show the wall-clock results. For the two heaviest benchmarks—event_sourcing (1.0 s) and game_rollback (1.9 s)—the difference is within measurement noise: -0.8% and $+1.2\%$

Benchmark	Description	Freezes	Thaws
alloc_churn	GC stress: tight alloc/drop loop	0	0
closure_heavy	Nested map/filter chains	0	0
event_sourcing	Frozen event log with periodic replay	200	600
freeze_thaw_cycle	Rapid freeze/thaw of structs	1,000	1,000
game_rollback	Periodic checkpoints with rollback	20	5
long_lived_crystal	Long-lived frozen values across GC	50	50
persistent_tree	Versioned collection with snapshots	200	21
undo_redo	Document editor undo/redo snapshots	75	36

Table 2: Benchmark suite characteristics. The first two benchmarks use no phase operations and serve as controls.

Benchmark	Regions (mean±sd)	No-Regions (mean±sd)	Δ
alloc_churn	0.031 ± 0.001 s	0.030 ± 0.001 s	−1.0%
closure_heavy	0.006 ± 0.000 s	0.006 ± 0.001 s	−5.1%
event_sourcing	0.992 ± 0.033 s	0.984 ± 0.014 s	−0.8%
freeze_thaw_cycle	0.007 ± 0.001 s	0.006 ± 0.000 s	−13.0%
game_rollback	1.861 ± 0.015 s	1.883 ± 0.023 s	+1.2%
long_lived_crystal	0.011 ± 0.000 s	0.011 ± 0.000 s	−2.3%
persistent_tree	0.220 ± 0.005 s	0.221 ± 0.013 s	+0.5%
undo_redo	0.118 ± 0.002 s	0.120 ± 0.001 s	+1.6%

Table 3: Wall-clock timing results. Δ shows the percentage change when using no-regions (positive = regions are faster). 20 runs per configuration.

respectively. The micro-benchmark freeze_thaw_cycle shows a 13% difference, but the absolute time is only 7 ms (sub-millisecond absolute difference), reflecting the per-operation cost of arena allocation.

6.4 Memory Results

Figure 7 and Table 4 present the memory results. Arena regions increase peak RSS proportionally to the volume of frozen data. The extreme case is freeze_thaw_cycle, which creates 1,000 persistent frozen structs, resulting in 231% RSS overhead. For realistic workloads (game_rollback, event_sourcing, undo_redo), the overhead ranges from 8% to 38%. The control benchmarks (alloc_churn, closure_heavy) show near-zero overhead, confirming that the dual-heap infrastructure is inactive when no phase operations occur.

6.5 Per-Operation Analysis

Table 5 breaks down the per-operation cost of freeze. With regions enabled, freeze performs a full deep-clone into arena pages, costing 8–70× more than the no-regions baseline (which merely flips the phase tag). However, the absolute cost is uniformly below 1 ms across all benchmarks, constituting less than 0.1% of total wall-clock time even in the heaviest workloads. This confirms that the arena migration overhead is dominated by interpreter evaluation time and does not affect

Benchmark	Regions RSS	No-Regions RSS	Region Data	RSS Overhead
alloc_churn	1,888 KB	1,888 KB	0 B	0.0%
closure_heavy	2,144 KB	2,128 KB	0 B	0.7%
event_sourcing	2,976 KB	2,160 KB	42,136 B	37.8%
freeze_thaw_cycle	5,888 KB	1,776 KB	151,200 B	231.5%
game_rollback	2,592 KB	2,400 KB	144,800 B	8.0%
long_lived_crystal	2,016 KB	1,792 KB	9,600 B	12.5%
persistent_tree	3,792 KB	2,992 KB	334,400 B	26.7%
undo_redo	2,608 KB	2,256 KB	85,200 B	15.6%

Table 4: Peak RSS and region data. RSS Overhead = (Regions − No-Regions) / No-Regions. The overhead correlates with region data volume: benchmarks with more frozen data show higher RSS.

Benchmark	Freezes	Freeze (Regions)	Freeze (No-Reg.)	Ratio
event_sourcing	200	0.160 ms	0.012 ms	13×
freeze_thaw_cycle	1,000	0.596 ms	0.027 ms	22×
game_rollback	20	0.516 ms	0.013 ms	40×
persistent_tree	200	0.261 ms	0.034 ms	8×
undo_redo	75	0.139 ms	0.002 ms	70×

Table 5: Freeze operation cost. With regions, freeze deep-clones into arena pages; without regions, it is essentially a phase-flag flip. Despite 8–70× per-operation slowdown, total freeze time is < 1 ms in all benchmarks.

overall performance.

6.6 Discussion of Results

The evaluation yields three key findings:

1. **Negligible wall-clock overhead.** The dual-heap architecture adds less than 2% wall-clock time for realistic workloads. The interpreter spends the vast majority of its time in evaluation (expression dispatch, environment lookups, deep cloning for value semantics), not in memory management.
2. **RSS proportional to frozen data.** Arena regions pre-allocate page-aligned memory for frozen values. The RSS overhead correlates directly with the volume of frozen data: workloads with many persistent frozen values (freeze_thaw_cycle) show high overhead, while workloads with few or no freezes show near-zero overhead.
3. **Freeze cost dominated by evaluation.** Individual freeze operations are 8–70× slower with regions (due to deep-cloning into arena pages), but the total freeze time is sub-millisecond in all benchmarks. The cost is amortized over the much larger evaluation budget.

These results demonstrate that the formal safety guarantees of the dual-heap architecture (heap separation, GC safety, bulk deallocation) come at negligible performance cost for real workloads.

7 Related Work

7.1 Immutability in Dynamic Languages

JavaScript `Object.freeze`. JavaScript's `Object.freeze` provides shallow immutability: only the object's own properties are frozen; nested objects remain mutable. Deep freezing requires manual recursion, and the mechanism is purely advisory—there is no compiler or runtime optimization based on frozen status. Lattice's `freeze` is deep by default and triggers migration to a dedicated memory region.

Clojure persistent data structures. Clojure [1] makes immutability the default through persistent data structures that share structure across versions. However, Clojure provides no mechanism for *controlled mutability*: transients offer a limited escape hatch, but the phase distinction is not first-class. Lattice inverts this design: mutability is the default (fluid), and immutability is explicitly requested (crystal), with bidirectional transitions.

Scala `val`/`var`. Scala distinguishes immutable (`val`) and mutable (`var`) bindings at the type level, but this distinction applies to the binding, not the value: a `val` holding a mutable collection still permits interior mutation. Lattice's phase tags apply to values themselves, ensuring deep transitivity.

Haskell. Haskell achieves immutability through purity, using monadic interfaces (`ST`, `IO`) for controlled mutation. This provides strong guarantees but requires threading state through types. Lattice targets the dynamic-language space where such type-level discipline is unavailable.

7.2 Ownership and Linear Types

Rust. Rust's ownership system [2] provides memory safety through affine types and borrow checking. Lattice's strict-mode consuming freeze ($\mathbf{freeze} : \tau^{\mathsf{fluid}} \multimap \tau^{\mathsf{crystal}}$) implements a localized form of affine resource management, but without requiring pervasive annotation. The key trade-off: Rust's guarantees are static and zero-cost; Lattice's are dynamic with deep-clone overhead.

Linear Haskell. Linear Haskell [3] extends Haskell's type system with linearity annotations. Like Rust, linearity is a type-level property; Lattice's is a runtime property. The advantage of runtime enforcement is that it works in a dynamically typed language without type annotations; the disadvantage is the overhead of deep cloning.

7.3 Region-Based Memory Management

Tofte-Talpin regions. The Tofte-Talpin region calculus [4] assigns each allocation to a lexically determined region, enabling stack-like deallocation. MLKit [5] implements this for Standard ML. Lattice's regions differ in that they are *phase-directed*: a value enters a region only when frozen, not based on lexical scope. This means regions serve a dual purpose—memory management and phase enforcement.

Cyclone. Cyclone [6] extends C with safe regions, fat pointers, and tagged unions. Cyclone's regions are type-checked at compile time; Lattice's are managed dynamically. Both achieve heap separation, but through different mechanisms: Cyclone via static analysis, Lattice via the freeze migration protocol.

Region-based garbage collection. Several systems combine regions with tracing GC. Qin et al. [7] describe a region-aware collector for Java. Lattice's approach is simpler: the GC's crystal fast-path (skip and record region ID) effectively partitions the heap, with bulk deallocation at region granularity replacing per-object collection for frozen data.

7.4 Value Semantics

Val. Val [8] (now Hylo) implements mutable value semantics with law-of-exclusivity enforcement, ensuring unique ownership of mutable state. Lattice shares the goal of preventing aliasing bugs but takes a different approach: rather than enforcing uniqueness at the type level, Lattice deep-clones on every read (value isolation) and provides explicit phase transitions for controlling mutability.

Swift value types. Swift's value types provide copy-on-write semantics for structs and enums. Lattice's deep-clone-on-read is similar in spirit but more aggressive: every variable read produces a full independent copy. Swift optimizes this with reference counting and CoW; Lattice does not yet implement CoW but the phase system provides a natural optimization point (crystal values could share backing storage).

7.5 GC and Immutability Optimizations

Several garbage collectors exploit immutability for optimization. OCaml's major heap [9] uses a write barrier that can skip scanning immutable records. Golang's GC [10] uses write barriers to track mutation. Java's ZGC [11] and Shenandoah GC [12] use colored pointers and barriers. Lattice's crystal fast-path is conceptually simpler: rather than instrumenting writes, it uses the phase tag to skip entire subgraphs during mark traversal.

8 Discussion and Future Work

8.1 Trade-offs

The Lattice phase system makes three deliberate trade-offs:

Deep clone cost. Value isolation via deep cloning on every read ensures safety but imposes overhead proportional to value size. For large arrays or deeply nested structures, this can be significant. In practice, the interpreter evaluation overhead dominates, as our benchmarks show.

RSS overhead of regions. Arena-allocated regions increase peak RSS proportionally to frozen data volume. This is the cost of maintaining pointer-disjoint heaps. For memory-constrained environments, the `--no-regions` flag provides a single-heap fallback that retains phase semantics without arena allocation.

Dynamic vs. static enforcement. Phase checking is primarily dynamic (with static analysis as a supplementary check), which means phase errors are caught at runtime rather than compile time. Strict mode provides the strongest guarantees, while casual mode offers a gentler learning curve.

8.2 Limitations

The current implementation has several limitations:

- **No copy-on-write.** Deep cloning is always eager. A CoW optimization for crystal values could eliminate redundant copies when the clone is never mutated.
- **Interpreter overhead.** As a tree-walking interpreter, Lattice's absolute performance is limited. A bytecode compiler or JIT would be needed for production use.
- **No concurrent GC.** The current mark-sweep collector is stop-the-world. Crystal values, being immutable and pointer-disjoint, are natural candidates for concurrent collection.
- **No generational collection.** The fluid heap uses a simple mark-sweep. Generational GC could improve throughput for short-lived fluid allocations.

8.3 Future Work

Copy-on-write sharing. Crystal values are deeply immutable, making them ideal candidates for CoW optimization. When a crystal value is thawed, the thawed copy could share backing storage with the original until mutation occurs.

Concurrent collection. The heap separation invariant guarantees that crystal regions are never modified after creation. This makes them safe to collect concurrently with mutator threads, as no write barrier is needed for crystal data.

Safe parallelism. The PC-Spawn-Strict rule already prevents fluid values from crossing thread boundaries. Combined with heap separation, this provides a foundation for safe data-parallel execution: crystal values can be freely shared across threads without locks or atomic operations.

JIT compilation. Phase tags provide optimization hints for a JIT compiler: crystal values can be inlined, their fields can be constant-folded, and arena-backed data can use more efficient access patterns.

WebAssembly target. Arena regions map naturally to WASM linear memory segments. A WASM backend could leverage the dual-heap architecture for efficient cross-language interop, with crystal values in shared memory and fluid values in instance-local memory.

9 Conclusion

We have presented the Lattice phase system, a runtime discipline for managing mutability in a dynamically typed language through first-class phase tags and explicit transition operations. The dual-heap memory architecture—a garbage-collected fluid heap paired with arena-based crystal regions—provides formal safety guarantees (six proved properties including heap separation and phase monotonicity) with negligible wall-clock overhead.

The key insight is that first-class immutability enables a clean separation of memory management concerns: mutable data is traced by a conventional GC, while immutable data resides in arena regions that offer bulk deallocation and GC fast-path skipping. This separation is not merely an optimization—it is a *safety mechanism* that prevents an entire class of GC-related bugs (dangling pointers to freed crystal data, accidental mutation through aliased references) and provides a foundation for future work in copy-on-write sharing, concurrent collection, and safe parallelism.

References

[1] R. Hickey. The Clojure programming language. In *DLS*, 2008.

[2] N. D. Matsakis and F. S. Klock. The Rust language. In *HILT*, 2014.

[3] J.-P. Bernardy, M. Boespflug, R. Newton, S. Peyton Jones, and A. Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *POPL*, 2018.

[4] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[5] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *PLDI*, 2002.

[6] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX ATC*, 2002.

[7] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[8] D. Racordon, D. Hess, and D. Abrahams. Implementation strategies for mutable value semantics. *Journal of Object Technology*, 2022.

[9] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system: Documentation and user's manual. INRIA, 2014.

[10] R. Hudson. Getting to Go: The journey of Go's garbage collector. GopherCon keynote, 2018.

[11] P. Lidén and S. Karlsson. ZGC: A scalable low-latency garbage collector. Oracle technical report, 2018.

[12] C. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *ISMM*, 2016.

Appendix A Full Proofs

This appendix contains complete proofs of the six safety properties stated in Section 4. We use the notation and definitions from Section 3.

Operation	Input Phase	Output Phase	Store Effect
freeze(e)	σ (any)	crystal	Migrate to region; free fluid ptrs
thaw(e)	σ (any)	fluid	Deep-clone into fluid heap
clone(e)	σ	σ	Deep-clone (same heap)
forge $\{\bar{s}\}$	—	crystal	Eval body, freeze result
flux $x = e$	σ	fluid	Tag as fluid
fix $x = e$	σ	crystal	Freeze and migrate to region
let $x = e$	σ	σ	Preserve inferred phase

Table 6: Phase transition operations summary.

A.1 Proof of Phase Monotonicity (Theorem 4.1)

Theorem A.1 (Phase Monotonicity — restated)**.** *If v^{crystal} is a crystal-tagged value, then no evaluation step can modify v or any sub-value of v in place. More precisely: let $\rho_{\text{raw}}(x) = v^{\mathsf{crystal}}$. Then for any assignment targeting x or a sub-path thereof, evaluation in strict mode produces* error*.*

Proof. We proceed by exhaustive case analysis on every evaluation rule that could modify a binding or its sub-values.

Part I. Direct identifier assignment ($x = e'$).

Case 1: Rule E-Assign*.* The guard requires $\mathcal{M} = \mathsf{strict} \Rightarrow \sigma_0 \neq \mathsf{crystal}$. Since $\sigma_0 = \mathsf{crystal}$, the guard fails and the rule is not applicable.

Case 2: Rule E-Assign-Crystal-Err*.* Both premises ($\mathcal{M} = \mathsf{strict}$ and $\rho_{\text{raw}}(x) = v_0^{\mathsf{crystal}}$) are satisfied. The conclusion is error.

Part II. Compound lvalue assignment.

Claim. If $\rho_{\text{raw}}(x) = v^{\mathsf{crystal}}$, then for any sub-path p starting at x, $\mathsf{resolve}(\rho, p)$ points to a crystal-phased sub-value.

Proof of Claim. By induction on path length. Base case: $\mathsf{resolve}(\rho, x) = \rho_{\text{ptr}}(x)$, which has phase crystal. Inductive case: by setphase (which recursively sets the phase during freeze), all sub-values of a crystal value are also crystal.

After resolution, the evaluator checks $\mathsf{phase}(\mathit{target}) = \mathsf{crystal}$ and produces error in strict mode.

Part III. Mutating method calls.

Array `.push()`, map `.set()`, and `.remove()` all check `value_is_crystal` before mutating. Crystal receivers produce errors.

Part IV. Static phase checking.

Rule PC-Assign requires $\Gamma(x) \neq \mathsf{crystal}$ in strict mode. Rule PC-Flux prevents crystal-to-fluid aliasing. These provide compile-time rejection in addition to runtime guards.

Synthesis. All mutation paths are blocked: direct assignment (Part I), compound lvalue (Part II), mutating methods (Part III), and phase demotion (Part IV). □

Remark A.2. *In casual mode, the runtime guards are conditional on $\mathcal{M} = \mathsf{strict}$. Phase monotonicity in casual mode relies on programmer discipline and partial static coverage.*

A.2 Proof of Value Isolation (Theorem 4.2)

The proof proceeds in three stages: precise definitions of pointer reachability, a key lemma establishing deep clone independence by structural induction, and the main theorem combining the lemma with the crystal exception.

A.2.1 Preliminary Definitions

Definition A.3 (Heap Address)**.** *A* heap address $a \in \mathsf{Addr}$ *is a pointer to a contiguous block of memory residing either in the fluid heap $\mathcal{F}$ or in an arena page of some crystal region $R \in \mathcal{R}$.*

Definition A.4 (Pointer Set)**.** *For a raw value $v \in \mathsf{Val}$, the* pointer set $\mathsf{ptrs}(v) \subseteq \mathsf{Addr}$ *is defined inductively:*

$$
\begin{aligned}
\mathsf{ptrs}(n) &= \emptyset && \textit{(integer, stored inline)}\\
\mathsf{ptrs}(r) &= \emptyset && \textit{(float, stored inline)}\\
\mathsf{ptrs}(b) &= \emptyset && \textit{(boolean, stored inline)}\\
\mathsf{ptrs}(()) &= \emptyset && \textit{(unit, no payload)}\\
\mathsf{ptrs}(s) &= \{\mathsf{buf}(s)\} && \textit{(string: buffer address)}\\
\mathsf{ptrs}([v_1, \ldots, v_k]) &= \{\mathsf{buf}(\mathit{elems})\} \cup \textstyle\bigcup_{i=1}^{k} \mathsf{ptrs}(v_i) && \textit{(array)}\\
\mathsf{ptrs}(\{s_1 : v_1, \ldots\}) &= \{\mathsf{buf}(\mathit{map})\} \cup \{\mathsf{buf}(\mathit{entries})\} && \textit{(map, plus keys}\\
&\quad \cup \textstyle\bigcup_i \big(\{\mathsf{buf}(s_i)\} \cup \{\mathsf{buf}(\mathit{vbox}_i)\} \cup \mathsf{ptrs}(v_i)\big) && \textit{and values)}\\
\mathsf{ptrs}(S\{f_1 : v_1, \ldots\}) &= \{\mathsf{buf}(\mathit{name})\} \cup \{\mathsf{buf}(\mathit{fnames})\} \cup \{\mathsf{buf}(\mathit{fvals})\} && \textit{(struct, plus}\\
&\quad \cup \textstyle\bigcup_i \big(\{\mathsf{buf}(f_i)\} \cup \mathsf{ptrs}(v_i)\big) && \textit{fields)}\\
\mathsf{ptrs}(\langle \bar{x}, e, \rho \rangle) &= \{\mathsf{buf}(\mathit{params})\} \cup \textstyle\bigcup_i \{\mathsf{buf}(x_i)\} \cup \mathsf{ptrs}_{\mathsf{env}}(\rho) && \textit{(closure)}
\end{aligned}
$$

where $\mathsf{buf}(\cdot)$ *denotes the address of a heap-allocated buffer and* $\mathsf{ptrs}_{\mathsf{env}}(\rho)$ *is the union of* $\mathsf{ptrs}(v)$ *for all values bound in ρ. The AST pointer e in a closure is excluded: it is a borrowed reference to the immutable parse tree, shared by all clones.*

Definition A.5 (Mutable Reachability)**.** *An address a is* mutably reachable *from v^σ if $a \in \mathsf{ptrs}(v)$ and either $\sigma \neq \mathsf{crystal}$ or a does not reside in a crystal region. We write $\mathsf{mreach}(v^\sigma)$ for this set.*

Definition A.6 (Shares Mutable State)**.** *Two tagged values $v_1^{\sigma_1}$ and $v_2^{\sigma_2}$* share mutable state *iff* $\mathsf{mreach}(v_1^{\sigma_1}) \cap \mathsf{mreach}(v_2^{\sigma_2}) \neq \emptyset$.

A.2.2 Deep Clone Independence Lemma

Lemma A.7 (Deep Clone Independence)**.** *For any tagged value v^σ, let $v'^\sigma = \mathsf{deepclone}(v^\sigma)$. Then:*

$$\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$$

Equivalently, $\mathsf{mreach}(v'^\sigma) \cap \mathsf{mreach}(v^\sigma) = \emptyset$.

Proof. By structural induction on the type of v.

Base cases.
Case $v = n$ (integer), $v = r$ (float), $v = b$ (boolean), $v = ()$ (unit): These are primitive types stored inline. $\mathsf{ptrs}(v) = \emptyset$, so $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$.
Case $v = s$ (string): Deep-clone invokes $\mathsf{lat\_strdup}(s)$, allocating a fresh buffer s'. $\mathsf{buf}(s') \neq \mathsf{buf}(s)$ by the allocator contract. $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$.

Inductive cases.
Case $v = [v_1, \ldots, v_k]$ (array): Deep-clone allocates a fresh element buffer via $\mathsf{lat\_alloc}$. For each i, $v_i' = \mathsf{deepclone}(v_i)$. By the inductive hypothesis on each element, $\mathsf{ptrs}(v_i') \cap \mathsf{ptrs}(v_i) \subseteq \mathsf{arena}(\mathcal{R})$. The fresh top-level buffer is distinct from the original. Thus $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$.
Case $v = \{s_1 : v_1, \ldots\}$ (map): Deep-clone allocates a fresh map structure, entries array, key strings (each via $\mathsf{lat\_strdup}$), and recursively clones each value. All top-level pointers are fresh. By the inductive hypothesis on sub-values, the result follows.
Case $v = S\{f_1 : v_1, \ldots\}$ (struct): Fresh buffers for the struct name, field-names array, field-values array, and each field name string. Recursive clones of field values satisfy the inductive hypothesis.
Case $v = \langle \bar{x}, e, \rho \rangle$ (closure): Fresh parameter-name array, fresh parameter name strings, and cloned captured environment via $\mathsf{env\_clone}(\rho)$, which deep-clones every binding. The body pointer e is shared but excluded from ptrs. Structural well-foundedness holds because Lattice environments cannot contain cyclic references. □

A.2.3 Main Theorem

Lemma A.8 (Crystal Protection)**.** *If $a \in \mathsf{arena}(\mathcal{R})$ is shared between v'^{σ} and v^{σ}, then a resides in a crystal region, and mutation at a is prevented by Phase Monotonicity.*

Proof. a belongs to crystal region R with identifier r. By E-Assign-Crystal-Err, any assignment targeting a crystal value produces error in strict mode. Deep-clone sets $\mathsf{region\_id} := \bot$ on the clone, but if the original had $\sigma = \mathsf{crystal}$, the clone inherits $\sigma = \mathsf{crystal}$ and Phase Monotonicity prevents mutation. If $\sigma \neq \mathsf{crystal}$, then neither value resides in a crystal region and the Independence Lemma gives $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$. □

Proof of Theorem 4.2. By E-Var, $v' = \mathsf{deepclone}(v)$ where $\rho_{\text{raw}}(x) = v^{\sigma}$. We must show $\mathsf{mreach}(v'^{\sigma}) \cap \mathsf{mreach}(v^{\sigma}) = \emptyset$.

By Lemma A.7, $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$. If the intersection is empty, isolation holds trivially. If some a is shared and $a \in \mathsf{arena}(\mathcal{R})$, then by Crystal Protection, a is excluded from mutable reachability. In all cases, $\mathsf{mreach}(v'^{\sigma}) \cap \mathsf{mreach}(v^{\sigma}) = \emptyset$. The store μ is unchanged by E-Var, confirming that variable reads are pure observations. □

A.3 Proof of Consuming Freeze (Theorem 4.3)

Proof. We structure the proof in four parts.

Part 1: Direct derivation from E-Freeze-Strict.

Assume $\mathcal{M} = \mathsf{strict}$ and evaluate $\mathbf{freeze}(x)$ in configuration $\langle \rho, (\mathcal{F}, \mathcal{R}), \mathbf{freeze}(x) \rangle$. The only applicable rule is E-Freeze-Strict:

E-Freeze-Strict

$$\frac{\mathcal{M} = \mathsf{strict} \qquad \rho_{\text{raw}}(x) = v^{\sigma} \qquad v'^{\mathsf{crystal}} = \mathsf{setphase}(v^{\sigma}, \mathsf{crystal}) \qquad (v''^{\mathsf{crystal}}, \mathcal{R}') = \mathsf{freeze\_region}(v'^{\mathsf{crystal}}, \mathcal{R})}{\langle \rho, (\mathcal{F}, \mathcal{R}), \mathbf{freeze}(x) \rangle \Downarrow \langle \rho \setminus x, (\mathcal{F}, \mathcal{R}'), v''^{\mathsf{crystal}} \rangle}$$

The resulting environment is $\rho' = \rho \setminus x$. By definition, $\mathrm{dom}(\rho \setminus x) = \mathrm{dom}(\rho) \setminus \{x\}$, so $x \notin \mathrm{dom}(\rho')$.

Part 2: Subsequent access to x produces an error.

The only rule for evaluating an identifier is E-Var, whose first premise requires $x \in \mathrm{dom}(\rho)$. Since $x \notin \mathrm{dom}(\rho')$, E-Var does not apply and evaluation produces: error("undefined variable 'x'").

Part 3: Contrast with casual mode.

In casual mode, the applicable rule is E-Freeze-Casual, which produces $\rho' = \rho[x \mapsto v''^{\mathsf{crystal}}]$—the binding persists with crystal phase. The critical difference:

- E-Freeze-Strict: $\rho' = \rho \setminus x$ (removal: binding consumed).
- E-Freeze-Casual: $\rho' = \rho[x \mapsto v''^{\mathsf{crystal}}]$ (update: binding persists as crystal).

Part 4: Static phase checker cooperation.

Rule PC-Freeze ensures in strict mode that **freeze** is never applied to an already-crystal value ($\sigma \neq \mathsf{crystal}$), preventing wasteful double-freeze and cooperating with the runtime to maintain the invariant: *in strict mode, freeze is applied exactly once to each fluid binding, after which the binding ceases to exist.*

Connection to linear type theory. Consuming freeze implements a localized form of affine resource management: **freeze** : $\tau^{\mathsf{fluid}} \multimap \tau^{\mathsf{crystal}}$, where $\multimap$ is the linear function arrow. Reads are unrestricted (contraction via deep clone), phase transitions are linear (the binding is consumed), and weakening is implicit (unused variables are garbage-collected). This achieves safety benefits of linear types—no fluid alias survives a freeze—without global annotation burden. □

A.4 Proof of Forge Soundness (Theorem 4.4)

Lemma A.9 (Setphase Totality)**.** $\mathsf{setphase}(v^\sigma, \sigma')$ *is total and produces phase* σ' *at every level.*

Proof. By structural induction on v. The recursion terminates on the finite structure and sets σ' at every node. □

Lemma A.10 (Freeze-Region Phase Preservation)**.** *If* v^{crystal}*, then* $\mathsf{freeze\_region}(v^{\mathsf{crystal}}, \mathcal{R}) = (v'^{\mathsf{crystal}}, \mathcal{R}')$.

Proof. Deep cloning preserves the phase tag; setting the region ID modifies only metadata. □

Proof of Theorem 4.4. By case analysis on the forge block's body evaluation:

Normal completion: body produces w^{σ_w}. Rule E-Forge applies setphase (forcing crystal by totality) then freeze_region (preserving crystal).

Early return: the return value undergoes the same freeze protocol; $\sigma = \mathsf{crystal}$.

Error, break, continue: no value produced; the theorem holds vacuously. □

Corollary A.11 (Forge Compositionality)**.** *Nested forge blocks produce crystal output at every level: the outer forge's* setphase *on an already-crystal value is idempotent.*

A.5 Proof of Heap Separation (Theorem 4.5)

We establish auxiliary definitions and five supporting lemmas before giving the main proof.

A.5.1 Auxiliary Definitions

Definition A.12 (Arena Pointer)**.** *A pointer a is an* arena pointer for region R *if there exists a page P_j in R's page list such that $P_j.data \leq a < P_j.data + P_j.cap$. We write $a \in \mathsf{arena}(R)$.*

Definition A.13 (Fluid-Registered Pointer)**.** *A pointer a is* fluid-registered *if $(a, n, m) \in \mathcal{F}$ for some size n and mark bit m. We write $a \in \mathrm{dom}(\mathcal{F})$.*

A.5.2 Supporting Lemmas

Lemma A.14 (Arena–Fluid Disjointness)**.** *For every region R managed by $\mathcal{R}$ and every pointer $a \in \mathsf{arena}(R)$: $a \notin \mathrm{dom}(\mathcal{F})$.*

Proof. Arena pages are allocated by direct `malloc` calls, bypassing fluid_alloc entirely. The system allocator guarantees that distinct live allocations return non-overlapping regions, so bump-pointer addresses within an arena page can never coincide with any address returned by fluid_alloc. □

Lemma A.15 (Arena Routing Completeness)**.** *When the global arena pointer g_{arena} is set to a region R, every call to* lat_alloc*,* lat_calloc*, or* lat_strdup *returns a pointer in* $\mathsf{arena}(R)$*.*

Proof. The arena check is the first branch in each allocator function. When $g_{arena} \neq \mathsf{null}$, execution unconditionally enters the arena path. No allocation can escape to fluid_alloc while the arena pointer is set. □

Lemma A.16 (Deep Clone Allocation Coverage)**.** $\mathsf{value\_deep\_clone}(v)$ *allocates every pointer in* $\mathsf{ptrs}(v')$ *(where v' is the clone) through calls to* lat_alloc*,* lat_calloc*, or* lat_strdup*.*

Proof. By structural induction on v. Scalar types have empty pointer sets. For strings, the clone calls lat_strdup. For arrays, the element buffer is allocated via lat_alloc and each element is recursively cloned. For structs, the name, field-names array, and field-values array are all allocated through lat_alloc or lat_strdup. For maps, when $g_{arena} \neq \mathsf{null}$, the clone builds map internals through lat_alloc/lat_calloc directly. For closures, environment cloning dispatches to env_clone_arena when the arena is active, which allocates all structures through the $\mathsf{lat\_}*$ family. □

Lemma A.17 (Region ID Completeness)**.** $\mathsf{set\_region\_id\_recursive}(v, r)$ *sets $v.region\_id := r$ on v and on every sub-value reachable from v, including array elements, struct field values, map entry values, and all values stored in closure captured environments.*

Proof. By structural induction on the value tree. For closures, the function calls set_region_id_env, which iterates all scopes and all bindings. □

Lemma A.18 (Value Free Removes Fluid Registrations)**.** *For a value v with $v.region\_id = \bot$,* $\mathsf{value\_free}(v)$ *calls* lat_free *on every pointer in* $\mathsf{ptrs}(v)$*, removing them from $\mathcal{F}$. For a value with $v.region\_id \neq \bot$,* $\mathsf{value\_free}(v)$ *is a no-op (the early return when $region\_id \neq \bot$ prevents freeing arena-backed pointers).*

A.5.3 Main Proof

Proof of Theorem 4.5. We prove that after the freeze migration protocol completes, the resulting crystal value v' (with region ID r) satisfies $\mathsf{ptrs}(v') \cap \mathrm{dom}(\mathcal{F}) = \emptyset$. The proof traces the seven steps of the protocol.

Step 1: Create a fresh region. $R \leftarrow \mathsf{region\_create}(\mathcal{R})$. Arena pages are allocated via direct `malloc`, disjoint from the fluid heap. $\mathcal{F}$ is unchanged.

Step 2: Set the global arena pointer. $g_{arena} \leftarrow R$. By Lemma A.15, all subsequent allocations go to $\mathsf{arena}(R)$. $\mathcal{F}$ unchanged.

Step 3: Deep-clone into R. $v' \leftarrow \mathsf{value\_deep\_clone}(v_0)$ with $g_{arena} = R$. By Lemma A.16, every pointer in $\mathsf{ptrs}(v')$ is allocated through lat_*. By Lemma A.15, every such pointer is in $\mathsf{arena}(R)$. By Lemma A.14, none appear in $\mathrm{dom}(\mathcal{F})$. $\mathcal{F}$ unchanged.

Step 4: Reset the arena pointer. $g_{arena} \leftarrow \mathsf{null}$. No allocations occur.

Step 5: Set region ID recursively. $\mathsf{set\_region\_id\_recursive}(v', r)$. By Lemma A.17, all sub-values carry region ID r, ensuring GC safety: the mark phase will record r and return immediately.

Step 6: Free the original. $\mathsf{value\_free}(v_0)$. By Lemma A.18, the old fluid allocations are removed from $\mathcal{F}$.

Step 7: Return the arena clone. The value slot is overwritten with the arena-backed clone.

Combining:

$$\begin{aligned} &\mathsf{ptrs}(v') \subseteq \mathsf{arena}(R) && \text{(Step 3)} \\ &\mathsf{arena}(R) \cap \mathrm{dom}(\mathcal{F}) = \emptyset && \text{(Lemma A.14)} \\ &\therefore\ \mathsf{ptrs}(v') \cap \mathrm{dom}(\mathcal{F}) = \emptyset \end{aligned}$$

Persistence. The invariant is preserved because: (1) arena pages remain allocated, so no future $\mathsf{fluid\_alloc}$ can return an address within them; (2) Phase Monotonicity (Theorem 4.1) prevents mutation of $\mathsf{ptrs}(v')$; (3) thaw produces independent copies (Theorem 4.6); (4) region collection frees entire pages atomically; (5) the GC mark phase respects the boundary via early return on crystal values with valid region IDs. The invariant is verified at runtime by $\mathsf{assert\_crystal\_not\_fluid}$ in debug builds. □

A.6 Proof of Thaw Independence (Theorem 4.6)

We use the pointer set notation from Definition A.7 and establish a key lemma before the main proof.

A.6.1 Key Lemma

Lemma A.19 (Deep Clone Produces Fresh Pointers)**.** *Let v^σ be a tagged value. Suppose* $\mathsf{g\_arena} = \mathsf{null}$ *at the time* $\mathsf{deepclone}(v^\sigma)$ *is invoked. Let $v'^\sigma = \mathsf{deepclone}(v^\sigma)$. Then:*

1. $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$ *(all pointers are fresh).*
2. *Every $a' \in \mathsf{ptrs}(v')$ satisfies $a' \in \mathrm{dom}(\mathcal{F})$ (all allocations go to the fluid heap).*
3. $v'.\mathsf{region\_id} = \perp$ *(the clone is not associated with any crystal region).*

Proof. By structural induction on v, following the same argument as Lemma A.7. Since $\mathsf{g\_arena} = \mathsf{null}$, all allocations route to $\mathsf{fluid\_alloc}$, producing fluid-heap pointers that are fresh by the allocator contract. Property (3) holds because $\mathsf{value\_deep\_clone}$ unconditionally sets $\mathsf{region\_id} := \perp$. □

Lemma A.20 ($\mathsf{setphase}$ Operates Only on Its Argument)**.** *The function* $\mathsf{setphase}(v^\sigma, \sigma')$ *modifies only the phase tags of v and its sub-values. It does not allocate or free any memory, follow any pointer not reachable from v, or modify any value not reachable from v.*

Proof. By inspection: $\mathsf{set\_phase\_recursive}$ performs only $v.phase := \sigma'$ assignments and recurses into structurally contained sub-values. □

A.6.2 Main Proof

Proof of Theorem 4.6. We proceed by case analysis on the two evaluation rules for **thaw**.

Case 1: E-Thaw-Expr.

Suppose e is not a bare identifier. The implementation evaluates e to obtain v^{σ}, then calls $\mathsf{value\_thaw}$, which deep-clones v and sets the phase to fluid.

Since no freeze-to-region is in progress, $\mathsf{g\_arena} = \mathsf{null}$. By Lemma A.19, the clone v' has $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$ and all pointers reside in $\mathrm{dom}(\mathcal{F})$. By Lemma A.20, the phase change does not touch the original. If the original was crystal in region R, R is unaffected.

Case 2: E-Thaw-Var.

Suppose the operand is a bare identifier x. Three values are in play:

(a) The *original binding* v^{σ} at $\rho_{\mathrm{raw}}(x)$.

(b) The *thawed value* v'^{fluid}, which replaces the binding.

(c) The *returned value* $v''^{\mathsf{fluid}} = \mathsf{deepclone}(v'^{\mathsf{fluid}})$.

The implementation: (1) $\mathsf{env\_get}$ deep-clones the binding (Theorem 4.2 gives independence from the original); (2) $\mathsf{value\_thaw}$ deep-clones again and sets phase to fluid; (3) a final $\mathsf{value\_deep\_clone}$ produces the return value; (4) $\mathsf{env\_set}$ stores the thawed value.

By Lemma A.19 applied at each clone step, and by the transitivity of allocator freshness:

$$\mathsf{ptrs}(v'') \cap \mathsf{ptrs}(\rho_{\mathrm{raw}}(x)) = \emptyset \qquad \text{and} \qquad \mathsf{ptrs}(v'') \cap \mathsf{ptrs}(v') = \emptyset$$

All pointers in v'' reside in $\mathrm{dom}(\mathcal{F})$. If the original binding was crystal with region r, the region $\mathcal{R}(r)$ is unmodified: thaw does not create, modify, or destroy any region.

Both cases establish pointer disjointness, phase correctness, and non-interference with crystal regions. □

Corollary A.21 (Crystal Safety Under Thaw)**.** *If* $\rho_{\mathrm{raw}}(x) = v^{\mathsf{crystal}}$ *with region* r*, then after evaluating* $\mathbf{thaw}(x)$*:*

1. *The crystal region* $\mathcal{R}(r)$ *remains valid and unmodified.*
2. *Any other binding with* $\mathsf{region\_id} = r$ *remains valid.*
3. *The thawed value can be freely mutated without affecting any crystal data.*

Proof. Properties (1) and (2) follow from the fact that thaw does not write to or deallocate any region. Property (3) follows from pointer disjointness: $\mathsf{ptrs}(v') \cap \mathsf{pages}(R) = \emptyset$ by Lemma A.19 and fluid heap residence. □

```
struct Event { kind: String, amount: Int, seq: Int }
struct Account { balance: Int, tx_count: Int }

fn apply_event(acct: Account, evt: Event) → Account {
    if evt.kind == "deposit" {
        return Account {
            balance: acct.balance + evt.amount,
            tx_count: acct.tx_count + 1
        }
    }
    if evt.kind == "withdraw" {
        return Account {
            balance: acct.balance - evt.amount,
            tx_count: acct.tx_count + 1
        }
    }
    return acct
}

fn replay(events: Array, count: Int) → Account {
    let acct = Account { balance: 0, tx_count: 0 }
    for i in 0..count {
        let evt = thaw(events[i])
        acct = apply_event(acct, evt)
    }
    return acct
}

fn main() {
    let event_log = []
    for i in 0..200 {
        let kind = "deposit"
        if i % 3 == 0 { kind = "withdraw" }
        let evt = Event { kind: kind,
            amount: (i % 50) + 1, seq: i }
        event_log.push(freeze(evt))
    }
    // Replay at periodic intervals
    for checkpoint in 0..5 {
        let count = (checkpoint + 1) * 40
        let state = replay(event_log, count)
    }
}
```

Listing 3: Event sourcing: frozen events form an append-only log.

PC-Freeze
$$\frac{\Gamma \vdash e : \sigma \qquad \mathcal{M} = \mathsf{strict} \Rightarrow \sigma \neq \mathsf{crystal}}{\Gamma \vdash \mathbf{freeze}(e) : \mathsf{crystal}}$$

PC-Assign
$$\frac{\Gamma \vdash e : \sigma_e \qquad \mathcal{M} = \mathsf{strict} \Rightarrow \Gamma(x) \neq \mathsf{crystal}}{\Gamma \vdash x = e \ \mathbf{ok}}$$

PC-Flux
$$\frac{\Gamma \vdash e : \sigma_e \qquad \mathcal{M} = \mathsf{strict} \Rightarrow \sigma_e \neq \mathsf{crystal}}{\Gamma \vdash \mathbf{flux}\ x = e\ \mathbf{ok} \qquad \Gamma' = \Gamma[x \mapsto \mathsf{fluid}]}$$

PC-Fix
$$\frac{\Gamma \vdash e : \sigma_e}{\Gamma \vdash \mathbf{fix}\ x = e\ \mathbf{ok} \qquad \Gamma' = \Gamma[x \mapsto \mathsf{crystal}]}$$

PC-Thaw
$$\frac{\Gamma \vdash e : \sigma \qquad \mathcal{M} = \mathsf{strict} \Rightarrow \sigma \neq \mathsf{fluid}}{\Gamma \vdash \mathbf{thaw}(e) : \mathsf{fluid}}$$

PC-Forge
$$\frac{\Gamma' = \Gamma \qquad \forall s_i \in \bar{s}.\ \Gamma' \vdash s_i\ \mathbf{ok}}{\Gamma \vdash \mathbf{forge}\ \{\ \bar{s}\ \} : \mathsf{crystal}}$$

PC-Spawn-Strict
$$\frac{\mathcal{M} = \mathsf{strict} \qquad x \in \mathrm{FV}(\bar{s}) \qquad \Gamma(x) = \mathsf{fluid}}{\Gamma \vdash \mathbf{spawn}\ \{\ \bar{s}\ \}\ \mathbf{ok}\,\dagger}$$

Figure 2: Selected static phase-checking rules. PC-Freeze ensures already-crystal values are not double-frozen in strict mode. PC-Assign prevents mutation of crystal bindings. PC-Flux prevents aliasing crystal data as fluid. PC-Spawn-Strict prevents fluid data from crossing thread boundaries.

E-Freeze-Strict
$$\frac{\mathcal{M} = \mathsf{strict} \qquad \rho_{\mathrm{raw}}(x) = v^{\sigma} \qquad v'^{\mathsf{crystal}} = \mathsf{setphase}(v^{\sigma}, \mathsf{crystal}) \qquad (v''^{\mathsf{crystal}}, \mathcal{R}') = \mathsf{freeze\_region}(v'^{\mathsf{crystal}}, \mathcal{R})}{\langle \rho, (\mathcal{F}, \mathcal{R}), \mathbf{freeze}(x) \rangle \Downarrow \langle \rho \setminus x, (\mathcal{F}, \mathcal{R}'), v''^{\mathsf{crystal}} \rangle}$$

E-Freeze-Casual
$$\frac{\mathcal{M} = \mathsf{casual} \qquad \rho_{\mathrm{raw}}(x) = v^{\sigma} \qquad v'^{\mathsf{crystal}} = \mathsf{setphase}(v^{\sigma}, \mathsf{crystal}) \qquad (v''^{\mathsf{crystal}}, \mathcal{R}') = \mathsf{freeze\_region}(v'^{\mathsf{crystal}}, \mathcal{R}) \qquad \rho' = \rho[x \mapsto v''^{\mathsf{crystal}}]}{\langle \rho, (\mathcal{F}, \mathcal{R}), \mathbf{freeze}(x) \rangle \Downarrow \langle \rho', (\mathcal{F}, \mathcal{R}'), \mathsf{deepclone}(v''^{\mathsf{crystal}}) \rangle}$$

Figure 3: Freeze semantics. In strict mode (E-Freeze-Strict), the binding is *consumed*: $\rho \setminus x$ removes x from the environment. In casual mode (E-Freeze-Casual), the binding is updated in place. Both modes migrate the value to an arena region.

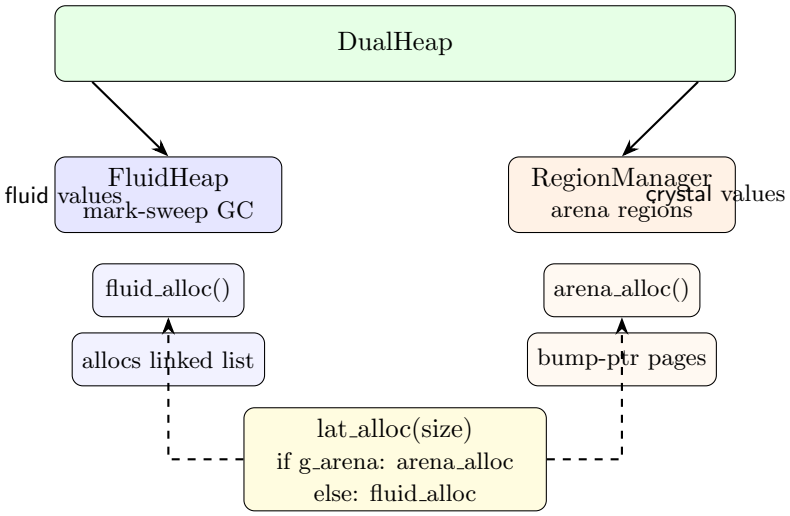

Figure 4: Dual-heap architecture. The `lat_alloc` router directs allocations to the arena during freeze migration and to the fluid heap otherwise. The two heaps have disjoint address spaces.

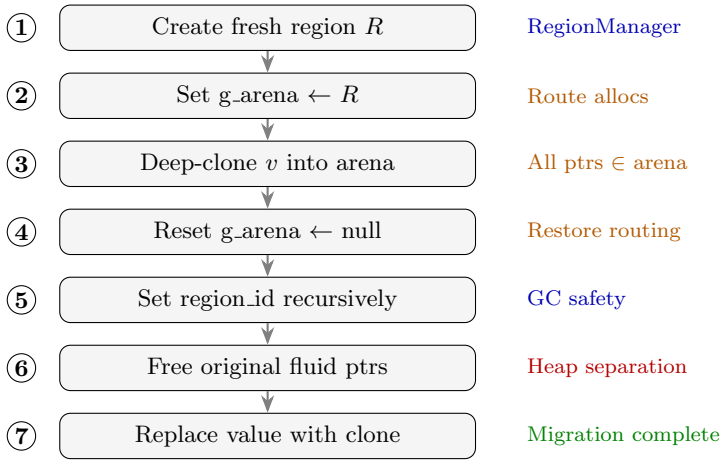

Figure 5: The seven-step freeze migration protocol. Steps 2–4 bracket the deep-clone operation, ensuring all allocations route through the arena. Step 6 removes original pointers from the fluid heap, completing heap separation.

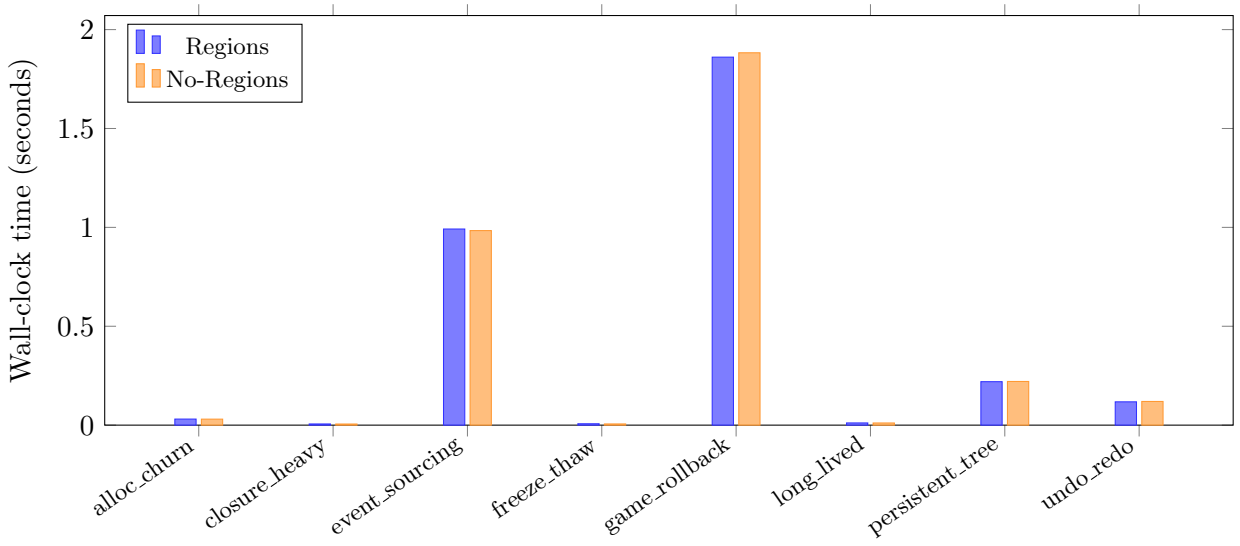

Figure 6: Wall-clock comparison across eight benchmarks (lower is better). Differences are within measurement noise for all workloads.

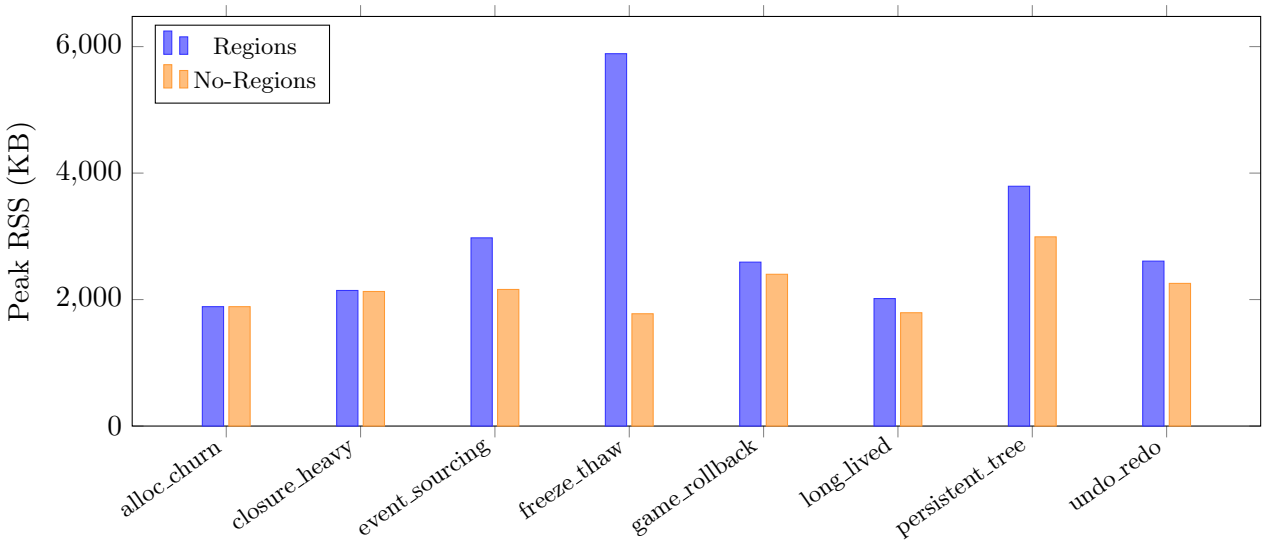

Figure 7: Peak RSS comparison (lower is better). Arena regions increase RSS proportionally to frozen data volume, from 0% (no freezes) to 70% (1,000 persistent frozen structs).