Formal Semantics of the Lattice Phase System

Alex Jokela
alex.c.jokela@gmail.com

February 2026

Abstract

We present formal operational semantics and static phase-checking rules for the *phase system* of the Lattice programming language. Lattice associates every runtime value with a *phase tag*—either fluid (mutable) or crystal (immutable)—and mediates transitions between phases through explicit **freeze**, **thaw**, and **clone** operations. The phase system is supported by a *dual-heap memory architecture*: a garbage-collected fluid heap for mutable data and an arena-based region store for frozen (crystal) data. We formalize the abstract syntax, a big-step operational semantics with store-passing, a static phase-checking judgment, and the memory model. We prove six key soundness properties: phase monotonicity, value isolation, consuming freeze semantics, forge soundness, heap separation, and thaw independence.

1 Phase Tags and Semantic Domains

Definition 1.1 (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 1.2 (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 1.3 (Environments and Stores)**.** *An* environment $\rho : \mathsf{Var} \rightharpoonup \mathsf{TVal}$ *maps variable names to tagged values. We write* $\rho[x \mapsto v^\sigma]$ *for extension and* $\rho \setminus x$ *for removal. Variable lookup produces a* deep clone*:*

$$\rho(x) = \mathsf{deepclone}(\rho_{\text{raw}}(x))$$

This ensures value isolation*: the caller receives an independent copy, and mutations to the copy do not affect the stored original.*

Definition 1.4 (Dual-Heap Store)**.** *The* store μ *is a disjoint union of two sub-stores:*

$$\mu = (\mathcal{F}, \mathcal{R}) \qquad \textit{where} \qquad \mathcal{F} \in \mathsf{FHeap} \qquad \textit{and} \qquad \mathcal{R} \in \mathsf{RStore} = \mathsf{RId} \rightharpoonup \mathsf{Region}$$

- $\mathcal{F}$ *(the* fluid heap*) is a set of tracked allocations managed by mark-sweep garbage collection. All* fluid- *and* $\bot$*-tagged values reside here.*
- $\mathcal{R}$ *(the* region store*) maps region identifiers to arena-allocated regions. Each region* R *is a sequence of pages with bump-pointer allocation. All* crystal*-tagged values reside in exactly one region.*

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)
	$\mid$	$\mathbf{if}\ e\ \{\ \bar{s}\ \}\ [\mathbf{else}\ \{\ \bar{s}\ \}]$	(conditional)
$s \in \mathsf{Stmt}$	::=	$d \mid x = e \mid e \mid \mathbf{return}\ e$	(statements)
	$\mid$	$\mathbf{for}\ x\ \mathbf{in}\ e\ \{\ \bar{s}\ \} \mid \mathbf{while}\ e\ \{\ \bar{s}\ \}$	(loops)

Lattice programs operate in one of two *modes*: $\mathcal{M} \in \{\mathsf{casual}, \mathsf{strict}\}$. The mode affects both the static checking rules and the runtime behavior of **freeze**.

3 Static Phase Checking

The static phase checker operates on the abstract syntax tree *before* evaluation. It maintains a *phase context* $\Gamma : \mathsf{Var} \rightharpoonup \mathsf{Phase}$ mapping variable names to their declared phases, and emits errors for phase violations.

3.1 Phase-Checking Judgment for Expressions

The judgment $\Gamma \vdash e : \sigma$ means "under context Γ, expression e has inferred phase σ."

$$\textsc{PC-Lit}\ \frac{c \text{ is a literal}}{\Gamma \vdash c : \bot} \qquad \textsc{PC-Var}\ \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \qquad \textsc{PC-BinOp}\ \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash e_1 \oplus e_2 : \bot}$$

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

$$\textsc{PC-Clone}\ \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathbf{clone}(e) : \sigma} \qquad \textsc{PC-Forge}\ \frac{\Gamma' = \Gamma \qquad \forall s_i \in \bar{s}.\ \Gamma' \vdash s_i\ \mathbf{ok}}{\Gamma \vdash \mathbf{forge}\ \{\ \bar{s}\ \} : \mathsf{crystal}}$$

$$\textsc{PC-FieldAcc}\ \frac{\Gamma \vdash e : \sigma \qquad \sigma' = \sigma \sqcup_{\mathsf{crystal}} \sigma \quad (\text{if } \sigma = \mathsf{crystal} \text{ then } \sigma' = \mathsf{crystal}, \text{ else } \sigma' = \sigma)}{\Gamma \vdash e.f : \sigma'} \qquad \textsc{PC-Closure}\ \frac{}{\Gamma \vdash \lvert\bar{x}\rvert\ \{\ \bar{s}\ \} : \bot}$$

3.2 Phase-Checking Judgment for Statements

The judgment $\Gamma \vdash s$ **ok** means "statement s is well-phased under Γ," and may extend Γ with new bindings.

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

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

$$\textsc{PC-Let-Casual}\quad \frac{\mathcal{M} = \mathsf{casual} \qquad \Gamma \vdash e : \sigma_e}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{ok} \qquad \Gamma' = \Gamma[x \mapsto \sigma_e]}$$

$$\textsc{PC-Let-Strict}\quad \frac{\mathcal{M} = \mathsf{strict}}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{ok}\,\dagger \quad \text{(error: use } \textbf{flux} \text{ or } \textbf{fix}\text{)}}$$

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

$$\textsc{PC-Spawn-Strict}\quad \frac{\mathcal{M} = \mathsf{strict} \qquad x \in \mathrm{FV}(\bar{s}) \qquad \Gamma(x) = \mathsf{fluid}}{\Gamma \vdash \textbf{spawn}\ \{\ \bar{s}\ \}\ \textbf{ok}\,\dagger \quad \text{(error: fluid across thread boundary)}}$$

The PC-Spawn-Strict rule prevents fluid (mutable) bindings from being captured across thread boundaries, enforcing that shared data must be crystal (immutable).

4 Big-Step Operational Semantics

We define a big-step (natural) semantics with store-passing. Configurations have the form $\langle \rho, \mu, e \rangle$ where ρ is the environment, $\mu = (\mathcal{F}, \mathcal{R})$ is the dual-heap store, and e is the expression to evaluate. Evaluation is written:

$$\langle \rho, \mu, e \rangle \Downarrow \langle \mu', v^\sigma \rangle$$

We omit the mode subscript when the rule applies uniformly to both modes.

4.1 Auxiliary Operations

Definition 4.1 (Deep Clone)**.** $\mathsf{deepclone}(v^\sigma)$ *produces a structurally identical value* v'^σ *where* v' *shares no mutable state with* v*. On compound values (arrays, structs, maps, closures), the operation recurses into all sub-values. Formally:*

$$\begin{aligned}
\mathsf{deepclone}(n^\sigma) &= n^\sigma && \textit{(integers, floats, bools: identity)}\\
\mathsf{deepclone}(s^\sigma) &= s'^\sigma && \textit{where } s' \textit{ is a fresh copy of string } s\\
\mathsf{deepclone}([v_1, \ldots, v_k]^\sigma) &= [\mathsf{deepclone}(v_1), \ldots, \mathsf{deepclone}(v_k)]^\sigma\\
\mathsf{deepclone}(\langle \bar{x}, e, \rho \rangle^\sigma) &= \langle \bar{x}, e, \mathsf{deepclone}(\rho) \rangle^\sigma
\end{aligned}$$

Definition 4.2 (Set Phase Recursively)**.** $\mathsf{setphase}(v^\sigma, \sigma')$ *produces* $v^{\sigma'}$ *with all sub-values also set to phase* σ'*. For compound values, the operation recurses into all fields, elements, and map entries.*

Definition 4.3 (Freeze-to-Region)**.** *The operation* $\mathsf{freeze\_region}(v^{\mathsf{crystal}}, \mathcal{R})$ *performs:*

1. *Allocate a fresh region* R *with identifier* r *in* $\mathcal{R}$*.*

2. *Deep-clone v into R's arena, producing v' with all pointers residing in R's arena pages.*
3. *Tag v' and all sub-values with region identifier r.*
4. *Deallocate the original fluid-heap pointers of v.*
5. *Return $(v'^{\mathsf{crystal}}, \mathcal{R}[r \mapsto R])$.*

4.2 Expression Evaluation Rules

4.2.1 Variables and Literals

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

Note that E-Var produces a *deep clone* of the stored value. This is the fundamental isolation property of Lattice: every variable read yields an independent copy, ensuring that no aliasing exists between the caller's copy and the environment's copy.

4.2.2 Phase Transition: Freeze

Strict mode (consuming freeze on identifiers). When $\mathcal{M} = \mathsf{strict}$ and **freeze** is applied to a variable name x, the binding is *consumed*—removed from the environment:

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

Casual mode (in-place freeze on identifiers). In casual mode, freezing a variable updates it in place and returns a deep clone of the frozen value:

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

Freeze on arbitrary expressions. When the operand is not a bare identifier, it is evaluated first:

$$\text{E-Freeze-Expr}\quad \frac{\langle \rho, \mu, e\rangle \Downarrow \langle \mu_1, v^{\sigma}\rangle \qquad v'^{\mathsf{crystal}} = \mathsf{setphase}(v^{\sigma}, \mathsf{crystal}) \qquad (v''^{\mathsf{crystal}}, \mathcal{R}') = \mathsf{freeze\_region}(v'^{\mathsf{crystal}}, \mathcal{R}_1)}{\langle \rho, \mu, \mathbf{freeze}(e)\rangle \Downarrow \langle (\mathcal{F}_1, \mathcal{R}'), v''^{\mathsf{crystal}}\rangle}$$

4.2.3 Phase Transition: Thaw

Thaw deep-clones a crystal value and sets the clone's phase to fluid. When applied to a variable, the binding is updated in place:

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

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

The notation $\mathsf{deepclone}(v^{\sigma})[\sigma := \mathsf{fluid}]$ denotes deep-cloning followed by recursively setting the phase to fluid on the clone, which matches the implementation where `value_thaw` calls `value_deep_clone` followed by `set_phase_recursive`.

4.2.4 Clone

Clone produces a deep copy preserving the original phase:

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

4.2.5 Forge Blocks

A **forge** block evaluates its body in a fresh scope and automatically freezes the result:

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

Here $\rho \oplus \{\}$ denotes pushing a fresh scope frame. The block is evaluated; its result (whether from the last expression or an early **return**) is frozen and migrated to a crystal region. Forge blocks are *compositional factories*: they guarantee their output is always crystal.

4.3 Statement Evaluation Rules

4.3.1 Binding Declarations

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

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

$$\text{E-Let-Casual}\quad \frac{\mathcal{M} = \mathsf{casual} \qquad \langle \rho, \mu, e \rangle \Downarrow \langle \mu', v^{\sigma} \rangle}{\langle \rho, \mu, \mathbf{let}\ x = e \rangle \Downarrow \langle \rho[x \mapsto v^{\sigma}], \mu', () \rangle} \qquad \text{E-Let-Strict}\quad \frac{\mathcal{M} = \mathsf{strict}}{\langle \rho, \mu, \mathbf{let}\ x = e \rangle \Downarrow \mathsf{error}(\text{“use flux or fix”})}$$

4.3.2 Assignment

Assignment operates directly on the store through *lvalue resolution*, which returns a mutable pointer into the environment rather than a deep clone:

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

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

4.3.3 Lvalue Resolution

Lvalue resolution $\mathsf{resolve}(\rho, l)$ walks chains of field accesses and index expressions to obtain a *direct mutable pointer* into the store. This contrasts with variable *reads*, which always deep-clone.

Definition 4.4 (Lvalue Resolution)**.**

$$\begin{aligned}
\mathsf{resolve}(\rho, x) &= \rho_{\text{ptr}}(x) && \textit{(direct pointer, no clone)}\\
\mathsf{resolve}(\rho, l.f) &= (\mathsf{resolve}(\rho, l)).\mathit{field}(f) && \textit{(struct field pointer)}\\
\mathsf{resolve}(\rho, l[e]) &= (\mathsf{resolve}(\rho, l)).\mathit{index}([\![e]\!]) && \textit{(array/map element pointer)}
\end{aligned}$$

This is the mechanism by which array element mutation (e.g., `a[i] = v`*) and struct field mutation operate* in-place *without cloning.*

5 Memory Model

The Lattice dual-heap memory model separates mutable and immutable data into distinct address spaces with different management strategies.

5.1 Fluid Heap

The fluid heap $\mathcal{F}$ is a linked list of tracked allocations:

$$\mathcal{F} = \{(p_i, n_i, m_i)\}_{i \in I}$$

where p_i is a pointer, n_i is the allocation size, and $m_i \in \{0, 1\}$ is the GC mark bit. Allocation is $O(1)$ (prepend to the list). The garbage collector uses a three-phase mark-sweep protocol:

1. **Unmark**: Set all $m_i := 0$.
2. **Mark**: Traverse all reachable values from root environments (including the current environment, saved caller environments for active closures, and a shadow stack for C-stack temporaries). For each reachable fluid pointer, set $m_i := 1$. For each reachable crystal value with a valid region ID, record the region ID in a reachable set.
3. **Sweep**: Free all allocations with $m_i = 0$. Concurrently, pass the reachable region set to region collection.

Crystal values are *never* swept by the fluid collector. If a value has $\sigma = \mathsf{crystal}$ and a valid region ID r, the mark phase merely records r as reachable and returns immediately, ensuring no crystal pointer is erroneously freed.

5.2 Region Store and Arena Allocation

Definition 5.1 (Crystal Region)**.** *A crystal region $R = (r, \varepsilon, P, n)$ consists of:*

- *A unique region identifier $r \in \mathsf{RId} = \mathbb{N}$.*
- *An epoch $\varepsilon \in \mathbb{N}$ (the epoch at creation time).*
- *A linked list of arena pages $P = [P_1, P_2, \ldots]$, each of size $|P_i| = 4096$ bytes (or larger for oversized allocations).*
- *Total bytes used n.*

Definition 5.2 (Arena Allocation)**.** *Given a region R with head page P_1:*

$$\mathsf{arena\_alloc}(R, k) = \begin{cases} P_1.data + P_1.used & \text{if } P_1.used + \lceil k \rceil_8 \leq P_1.cap \\ P'.data & \text{otherwise, where } P' \text{ is a fresh page prepended to } R \end{cases}$$

where $\lceil k \rceil_8$ denotes 8-byte alignment. Arena allocation is $O(1)$ amortized (bump-pointer within a page, $O(1)$ page allocation).

Definition 5.3 (Region Collection)**.** *Given the set of reachable region IDs $\mathcal{S} \subseteq \mathsf{RId}$ collected during the mark phase:*

$$\mathsf{region\_collect}(\mathcal{R}, \mathcal{S}) = \{r \mapsto R \mid (r \mapsto R) \in \mathcal{R},\ r \in \mathcal{S}\}$$

All regions $r \notin \mathcal{S}$ are freed in $O(|P|)$ time (free each page in the linked list). This provides bulk deallocation: all data in an unreachable region is freed at once without per-object overhead.

5.3 The Freeze Migration Protocol

The central operation linking the two heaps is the *freeze migration*:

$$\mathsf{freeze\_region}(v^{\mathsf{crystal}}, (\mathcal{F}, \mathcal{R})) :$$

1. Create a fresh region R in $\mathcal{R}$ with $r = \mathsf{next\_id}$.
2. Set the global arena pointer to R (so all allocation functions route to R's arena).
3. Deep-clone v into R: $v' = \mathsf{deepclone}(v)$, where all allocations in the clone go through $\mathsf{arena\_alloc}(R, \cdot)$.
4. Reset the arena pointer to null.
5. Recursively set $\mathsf{region\_id} := r$ on v' and all sub-values.
6. Free the original fluid-heap pointers of v via $\mathsf{value\_free}$.
7. Return v'^{crystal} with updated $\mathcal{R}$.

After migration, the crystal value v' has completely independent pointers from $\mathcal{F}$. This is verified by the debug assertion ASSERT-CRYSTAL-NOT-FLUID: no pointer reachable from a crystal value with a valid region ID may appear in the fluid allocation list.

6 Properties

We state the key invariants and properties of the phase system. Complete proofs are provided in Sections 8 through 13.

Theorem 6.1 (Phase Monotonicity)**.** *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 $\rho_{\text{raw}}(x) = v^{\mathsf{crystal}}$ for some x. Then for any statement $x = e'$ or lvalue assignment targeting a sub-path of x, evaluation in strict mode produces error*, and evaluation in casual mode is undefined behavior (rejected by the static checker when possible).*

Theorem 6.2 (Value Isolation)**.** *Variable reads produce independent copies: if* $\langle \rho, \mu, x \rangle \Downarrow \langle \mu, v'^{\sigma} \rangle$*, then* v' *shares no mutable state with* $\rho_{\text{raw}}(x)$*.*

Formally, for any address a *reachable from* v' *and any address* b *reachable from* $\rho_{\text{raw}}(x)$*:* $a \neq b$ *(unless* a *is in a crystal region, in which case mutation is prevented by phase monotonicity).*

Theorem 6.3 (Consuming Freeze)**.** *In strict mode (*$\mathcal{M} = \mathsf{strict}$*), after evaluating* $\mathbf{freeze}(x)$*, the variable* x *is no longer bound in the environment:*

$$\langle \rho, \mu, \mathbf{freeze}(x) \rangle \Downarrow \langle \rho', \mu', v^{\mathsf{crystal}} \rangle \quad \Longrightarrow \quad x \notin \mathrm{dom}(\rho')$$

This prevents use-after-freeze of the original mutable data, providing a lightweight form of linear resource management.

Theorem 6.4 (Forge Soundness)**.** *The result of a* **forge** *block is always crystal-phased:*

$$\langle \rho, \mu, \mathbf{forge}\ \{\ \bar{s}\ \} \rangle \Downarrow \langle \mu', v^{\sigma} \rangle \quad \Longrightarrow \quad \sigma = \mathsf{crystal}$$

Theorem 6.5 (Heap Separation)**.** *No pointer reachable from a crystal value with a valid region ID* $r \neq \bot$ *appears in the fluid heap's allocation list:*

$$\forall\, v^{\mathsf{crystal}} \text{ with region } r.\ \forall\, a \in \mathsf{ptrs}(v).\ a \notin \mathrm{dom}(\mathcal{F})$$

This ensures GC safety: the fluid sweep can never free an arena-backed crystal pointer.

Theorem 6.6 (Thaw Independence)**.** *The value produced by* $\mathbf{thaw}(e)$ *is a fresh, deep-cloned copy of the original with phase set to* fluid*. It shares no state with the source:*

$$\langle \rho, \mu, \mathbf{thaw}(e) \rangle \Downarrow \langle \mu', v'^{\mathsf{fluid}} \rangle \quad \Longrightarrow \quad \mathsf{ptrs}(v') \cap \mathsf{ptrs}([\![e]\!]) = \emptyset$$

In particular, thawing a crystal value does not invalidate the crystal region; the thawed copy resides in the fluid heap.

Proposition 6.7 (Region Bulk Deallocation)**.** *When a crystal region* R *becomes unreachable, all data within* R *is freed in time* $O(|P_R|)$ *where* $|P_R|$ *is the number of arena pages, independent of the number of individual values stored in the region. This provides constant-factor overhead per page rather than per value.*

7 Summary of Phase Transitions

The following table summarizes the phase transition operations and their effects on the store:

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

The read/write asymmetry is a defining characteristic of Lattice's memory model: *reads* always deep-clone (via `env_get`), ensuring value-semantics isolation, while *writes* resolve lvalue paths to direct pointers (via `resolve_lvalue`), enabling efficient in-place mutation of fluid data.

8 Proof of Phase Monotonicity

We now give a complete proof of Theorem 6.1 (Phase Monotonicity). We restate it here for convenience.

Theorem 8.1 (Phase Monotonicity — Theorem 6.1 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 $\rho_{\text{raw}}(x) = v^{\mathsf{crystal}}$ for some variable x. Then for any statement $x = e'$ or lvalue assignment targeting a sub-path of x (i.e., $x.f_1.\cdots.f_k = e'$ or $x[i_1]\cdots[i_k] = e'$, or any mixed chain thereof), evaluation in strict mode ($\mathcal{M} = \mathsf{strict}$) produces error*. In casual mode ($\mathcal{M} = \mathsf{casual}$), the static phase checker rejects the program before evaluation whenever the binding is statically known to be crystal.*

Proof. We proceed by exhaustive case analysis on every evaluation rule and auxiliary operation that could, in principle, modify a binding $\rho(x)$ or a sub-value thereof. In each case we show that a crystal phase tag causes the evaluator (in strict mode) to produce an error, and that the static phase checker (in strict mode) independently rejects the offending statement at analysis time.

The proof is organized into four parts: (I) direct identifier assignment, (II) compound lvalue assignment through sub-paths, (III) mutating method calls, and (IV) static phase checking.

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

The only evaluation rules that can modify the top-level binding of a variable x through assignment are E-Assign and E-Assign-Crystal-Err. We show that when $\rho_{\text{raw}}(x) = v^{\mathsf{crystal}}$ and $\mathcal{M} = \mathsf{strict}$, the assignment necessarily fails.

Case 1: Rule E-Assign*.* This rule has the premises:

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

The third premise is a *guard*: it requires that if we are in strict mode, then $\sigma_0 \neq \mathsf{crystal}$. By hypothesis, $\rho_{\text{raw}}(x) = v^{\mathsf{crystal}}$, so $\sigma_0 = \mathsf{crystal}$. In strict mode, the guard $\sigma_0 \neq \mathsf{crystal}$ fails. Therefore rule E-Assign is *not applicable*, and this case cannot produce a successful update.

Case 2: Rule E-Assign-Crystal-Err*.* This rule has the premises:

$$\mathcal{M} = \mathsf{strict} \qquad \rho_{\text{raw}}(x) = v_0^{\mathsf{crystal}}$$

Both premises are satisfied by our hypotheses ($\mathcal{M} = \mathsf{strict}$ and $\sigma_0 = \mathsf{crystal}$). The conclusion is:

$$\langle \rho, \mu, x = e' \rangle \Downarrow \mathsf{error}(\text{“cannot assign to crystal binding”})$$

Hence, in strict mode, any attempt at direct assignment to a crystal binding produces an error.

Implementation correspondence. In the C implementation (`eval.c`, lines 1598–1611), the `STMT_ASSIGN` handler for identifier targets performs exactly this check. It calls `env_get` to retrieve the existing binding, then tests `value_is_crystal(&existing)`. If the value's phase tag is `VTAG_CRYSTAL` and the evaluator mode is `MODE_STRICT`, the function returns an error via `eval_err`, matching E-Assign-Crystal-Err.

Part II. Compound lvalue assignment ($x.f = e'$, $x[i] = e'$, and mixed chains).

For assignments to sub-paths of x (e.g., $x.f = e'$, $x[i] = e'$, $x.f[i].g = e'$), evaluation uses the resolve (lvalue resolution) mechanism defined in Section 4. We must show that every such path that terminates inside a crystal value is rejected.

The evaluator handles compound lvalue assignment in two stages:

1. *Lvalue resolution*: $\mathsf{resolve}(\rho, l)$ traverses the path l and returns a direct mutable pointer to the target sub-value.
2. *Phase guard*: after resolution, the evaluator checks whether the resolved target is crystal-phased, and if so (in strict mode), produces an error.

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

Proof of Claim. We proceed by induction on the length of the path p.

- *Base case* ($p = x$). Then $\mathsf{resolve}(\rho, x) = \rho_{\text{ptr}}(x)$, which points directly to the stored value v^{crystal}. The phase is crystal.
- *Inductive case: field access* ($p = q.f$). By the induction hypothesis, $\mathsf{resolve}(\rho, q)$ points to a value w^{crystal}. Since w is crystal-tagged and all sub-values of a crystal value are also crystal (by the definition of setphase in Section 4, which recursively sets the phase on all fields, elements, and map entries during the freeze operation), the field $w.f$ is itself crystal-tagged. Thus $\mathsf{resolve}(\rho, q.f) = (\mathsf{resolve}(\rho, q)).\mathit{field}(f)$ points to a crystal-phased sub-value.
- *Inductive case: index access* ($p = q[i]$). By the same reasoning as the field access case, $\mathsf{resolve}(\rho, q)$ points to a crystal value, and since setphase recurses into array elements and map entries, the element at index i is also crystal-phased. Thus $\mathsf{resolve}(\rho, q[i])$ points to a crystal-phased sub-value.

This completes the induction. In every case, the resolved target carries phase crystal. $\square_{\text{Claim}}$

Now, after lvalue resolution returns a pointer *target* to the sub-value, the evaluator performs the following check (corresponding to lines 1628–1631 of `eval.c`):

$$\mathcal{M} = \mathsf{strict} \ \wedge \ \mathsf{phase}(\mathit{target}) = \mathsf{crystal} \quad \Longrightarrow \quad \mathsf{error}(\text{“cannot assign to crystal value”})$$

By the Claim, whenever $\rho_{\text{raw}}(x) = v^{\mathsf{crystal}}$ and we resolve any sub-path of x, the target is crystal-phased. In strict mode, the guard fires and produces an error. Therefore, no compound lvalue assignment into a crystal binding can succeed.

Part III. Mutating method calls on crystal values.

In addition to assignment statements, in-place mutation can occur through method calls such as `.push()`, `.set()`, and `.remove()`. We must show that these are also rejected on crystal values.

- *Array* `.push()`: The evaluator checks `value_is_crystal(&existing)` before performing the push operation (`eval.c`, line 1078). If the array is crystal, it returns $\mathsf{error}(\text{“cannot push to a crystal array”})$.
- *Map* `.set()` *and* `.remove()`: These methods use `resolve_lvalue` to obtain a mutable pointer to the map. Since the map is a sub-value of a crystal binding (or is itself crystal), the resolved pointer carries phase crystal. The phase system's invariant is that crystal values are only produced by `fix`, `freeze`, or `forge`, all of which set the phase recursively, so the value itself carries the crystal tag. A well-phased program in strict mode cannot hold a mutable reference to a crystal map.

Part IV. Static phase checking rejects crystal assignment at analysis time.

The runtime checks described in Parts I–III serve as a dynamic safety net. In strict mode, the *static* phase checker provides an additional, *earlier* line of defense by rejecting crystal assignments before evaluation begins.

Rule PC-Assign. The static phase-checking judgment for assignment statements is:

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

The second premise requires that in strict mode, the phase context Γ does not map x to crystal. If x was declared with **fix** (which records $\Gamma(x) = \mathsf{crystal}$ per rule PC-Fix), then $\Gamma(x) = \mathsf{crystal}$ and the premise $\Gamma(x) \neq \mathsf{crystal}$ fails. The static checker therefore rejects the statement $x = e'$ with a phase error.

Binding declarations also prevent crystal-to-fluid demotion. Rule PC-Flux additionally prevents binding a crystal expression with **flux**:

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

In strict mode, if e has phase crystal, the premise $\sigma_e \neq \mathsf{crystal}$ fails. This prevents the creation of a fluid-tagged alias of crystal data, which would circumvent the monotonicity invariant.

Synthesis.

We have shown that every mechanism by which a crystal-tagged binding or its sub-values could be modified in place is blocked by the Lattice phase system:

1. *Direct assignment* ($x = e'$): blocked by E-Assign-Crystal-Err at runtime and by PC-Assign at static analysis time (Parts I and IV).
2. *Compound lvalue assignment* ($x.f = e'$, $x[i] = e'$, and deeper chains): blocked by the post-resolution crystal guard, which relies on the recursive phase invariant established by setphase (Part II).
3. *Mutating method calls* (`.push()`, etc.): blocked by explicit crystal checks on the receiver value (Part III).
4. *Phase demotion* (**flux** $y = x$ where x is crystal): blocked by PC-Flux and the corresponding runtime check, preventing the creation of fluid aliases that could bypass the invariant (Part IV).

Since these four categories exhaust all mutation paths in the Lattice evaluation semantics, no evaluation step can modify a crystal-tagged value or any of its sub-values in place. This completes the proof. □

Remark 8.2 (Casual mode)**.** *In casual mode ($\mathcal{M} = \mathsf{casual}$), the runtime guards in* E-Assign *and the lvalue post-resolution check do not enforce the crystal constraint (the guards are conditional on $\mathcal{M} = \mathsf{strict}$). Phase monotonicity in casual mode therefore relies entirely on programmer discipline and on partial coverage from the static phase checker (which still issues warnings for statically detectable crystal assignments). The formal guarantee of Theorem 6.1 is thus strongest in strict mode; in casual mode it holds only to the extent that the static checker can track phases—a deliberate design trade-off that provides a gentler on-ramp for new users while preserving full safety for production (strict-mode) code.*

Remark 8.3 (Relationship to heap separation)**.** *Phase monotonicity is strengthened by the heap separation property (Theorem 6.5). Crystal values reside in arena-allocated regions with no pointers in the fluid heap. Even if a bug were to bypass the phase tag check, the crystal data's arena-backed pointers are never freed by the fluid-heap garbage collector, providing a second line of defense against*

accidental mutation through use-after-free on the fluid heap. The debug assertion ASSERT-CRYSTAL-NOT-FLUID *(implemented as* `assert_crystal_not_fluid` *in* `eval.c`*) validates this invariant after every garbage collection cycle.*

9 Proof of Value Isolation (Theorem 6.2)

We now give a rigorous proof of Theorem 6.2, restated here for convenience.

Theorem 6.2 (Value Isolation). *Variable reads produce independent copies: if* $\langle \rho, \mu, x \rangle \Downarrow \langle \mu, v'^{\sigma} \rangle$*, then* v' *shares no mutable state with* $\rho_{\text{raw}}(x)$*. Formally, for any address* a *reachable from* v' *and any address* b *reachable from* $\rho_{\text{raw}}(x)$*:* $a \neq b$ *(unless* a *resides in a crystal region, in which case mutation is prevented by Theorem 6.1, Phase Monotonicity).*

The proof proceeds in three stages. First, we make precise the notion of *sharing mutable state* by defining pointer reachability. Second, we establish a key lemma: that deepclone produces structurally independent values, proved by structural induction on the value type. Third, we combine the lemma with the E-VAR rule and the crystal exception to discharge the theorem.

9.1 Preliminary Definitions

Definition 9.1 (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* $R \in \mathcal{R}$*. In the implementation, addresses correspond to the pointers returned by* `fluid_alloc`*,* `arena_alloc`*,* `malloc`*, or their* `calloc`*/*`strdup` *variants.*

Definition 9.2 (Pointer Set)**.** *For a raw value* $v \in \mathsf{Val}$*, the* pointer set $\mathsf{ptrs}(v) \subseteq \mathsf{Addr}$ *is defined inductively:*

$$\mathsf{ptrs}(n) = \emptyset \qquad (n \in \mathbb{Z}\text{: integ}$$

$$\mathsf{ptrs}(r) = \emptyset \qquad (r \in \mathbb{R}\text{: float,}$$

$$\mathsf{ptrs}(b) = \emptyset \qquad (b \in \mathbb{B}\text{: boole}$$

$$\mathsf{ptrs}(()) = \emptyset \qquad \textit{(unit, no pay}$$

$$\mathsf{ptrs}(\mathit{rng}) = \emptyset \qquad \textit{(range, two i}$$

$$\mathsf{ptrs}(s) = \{\mathsf{buf}(s)\} \qquad \textit{(string: addr}$$

$$\mathsf{ptrs}([v_1, \ldots, v_k]) = \{\mathsf{buf}(\mathit{elems})\} \cup \bigcup_{i=1}^{k} \mathsf{ptrs}(v_i) \qquad \textit{(array: eleme}$$

$$\mathsf{ptrs}(\{s_1 : v_1, \ldots, s_k : v_k\}) = \{\mathsf{buf}(\mathit{map})\} \cup \{\mathsf{buf}(\mathit{entries})\} \cup \bigcup_{i=1}^{k} \big(\{\mathsf{buf}(s_i)\} \cup \{\mathsf{buf}(\mathit{vbox}_i)\} \cup \mathsf{ptrs}(v_i)\big) \qquad \textit{(map)}$$

$$\mathsf{ptrs}(S\{f_1 : v_1, \ldots, f_k : v_k\}) = \{\mathsf{buf}(\mathit{name})\} \cup \{\mathsf{buf}(\mathit{fnames})\} \cup \{\mathsf{buf}(\mathit{fvals})\} \cup \bigcup_{i=1}^{k} \big(\{\mathsf{buf}(f_i)\} \cup \mathsf{ptrs}(v_i)\big) \qquad \textit{(struct)}$$

$$\mathsf{ptrs}(\langle \bar{x}, e, \rho \rangle) = \{\mathsf{buf}(\mathit{params})\} \cup \bigcup_{i=1}^{|\bar{x}|} \{\mathsf{buf}(x_i)\} \cup \mathsf{ptrs}_{\mathsf{env}}(\rho) \qquad \textit{(closure)}$$

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* v *bound in* ρ*, together with the addresses of the environment's scope arrays and hash-map backing stores.*

Note that the AST pointer e *in a closure is* excluded *from* ptrs*: the body is a borrowed reference to the immutable parse tree, shared by all clones, and is never mutated during evaluation.*

Definition 9.3 (Mutable Reachability)**.** *An address* a *is* mutably reachable *from a tagged value* v^σ *if:*

1. $a \in \mathsf{ptrs}(v)$*, and*
2. *either* $\sigma \neq \mathsf{crystal}$*, or* a *does not reside in a crystal region (i.e., the value's* $\mathsf{region\_id} = \bot$*).*

We write $\mathsf{mreach}(v^\sigma)$ *for the set of mutably reachable addresses from* v^σ*. Crystal-region addresses are excluded because they are protected by Phase Monotonicity (Theorem 6.1).*

Definition 9.4 (Shares Mutable State)**.** *Two tagged values* $v_1^{\sigma_1}$ *and* $v_2^{\sigma_2}$ share mutable state *if and only if*

$$\mathsf{mreach}(v_1^{\sigma_1}) \cap \mathsf{mreach}(v_2^{\sigma_2}) \neq \emptyset.$$

That is, there exists some heap address that is mutably reachable from both values. If one value could write through such an address, the effect would be visible from the other.

9.2 Deep Clone Independence Lemma

Lemma 9.5 (Deep Clone Independence)**.** *For any tagged value* v^σ *stored in the environment, let* $v'^\sigma = \mathsf{deepclone}(v^\sigma)$*. Then:*

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

where $\mathsf{arena}(\mathcal{R})$ *is the set of all addresses residing in crystal region arenas. That is, every non-arena pointer in* v' *is fresh—distinct from every pointer in* v*.*

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

Proof. We proceed by structural induction on the type of v.

Base cases.

Case $v = n$ **(integer)**, $v = r$ **(float)**, $v = b$ **(boolean)**, $v = ()$ **(unit)**, $v = (s, e)$ **(range)**:

These are *primitive* value types stored entirely inline within the `LatValue` struct. No heap allocation is performed. Since $\mathsf{ptrs}(v) = \emptyset$, we have $\mathsf{ptrs}(v') = \emptyset$, and hence $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset \subseteq \mathsf{arena}(\mathcal{R})$.

Case $v = s$ **(string)**:

The deep-clone operation invokes $\mathsf{lat\_strdup}(s)$, which allocates a *fresh* buffer s' via lat_alloc. In all cases, $\mathsf{buf}(s')$ is a freshly allocated address, distinct from $\mathsf{buf}(s)$. Therefore $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$.

Inductive cases.

Case $v = [v_1, \ldots, v_k]$ **(array)**:

The deep-clone operation allocates a fresh element buffer via 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 $v = \{s_1 : v_1, \ldots, s_k : v_k\}$ **(map)**:

The deep-clone allocates a fresh `LatMap` structure, a fresh entries array, fresh key strings, and recursively clones each value. All top-level pointers are freshly allocated and therefore distinct

from any pointer in v. The recursive sub-value clones satisfy the inductive hypothesis. Thus $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$.

Case $v = S\{f_1 : v_1, \ldots, f_k : v_k\}$ **(struct)**:

The deep-clone allocates fresh buffers for the struct name, field-names array, field-values array, and each field name string, and recursively clones each field value. Every top-level pointer is fresh by the allocator contract. Each recursive clone satisfies the inductive hypothesis. Thus $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$.

Case $v = \langle \bar{x}, e, \rho_c \rangle$ **(closure)**:

The deep-clone allocates a fresh parameter-name array, fresh parameter name strings, and clones the captured environment via $\mathsf{env\_clone}(\rho_c)$, which deep-clones every binding in every scope. The body pointer e is shared but excluded from ptrs. By the inductive hypothesis applied to each value in the captured environment, every cloned binding has fresh pointers. The structural well-foundedness is guaranteed because Lattice environments cannot contain cyclic references. Thus $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$.

This completes the structural induction over all value types. □

9.3 Proof of Value Isolation

We additionally require the following observation about crystal addresses.

Lemma 9.6 (Crystal Protection)**.** *If an address $a \in \mathsf{arena}(\mathcal{R})$ is shared between v'^{σ} and $\rho_{\text{raw}}(x) = v^{\sigma}$, then a resides in a crystal region, and any attempt to mutate the value at a is prevented by Phase Monotonicity (Theorem 6.1).*

Proof. Suppose $a \in \mathsf{ptrs}(v') \cap \mathsf{ptrs}(v)$ and $a \in \mathsf{arena}(\mathcal{R})$. Then a belongs to some crystal region R with identifier r. By the E-Assign-Crystal-Err rule, any assignment targeting a crystal-tagged value produces error in strict mode. Furthermore, the deep-clone operation sets $\mathsf{region\_id} := \bot$ on the cloned value, meaning the clone v' is not tagged as belonging to any crystal region. If the original had $\sigma = \mathsf{crystal}$, then the clone inherits $\sigma = \mathsf{crystal}$ and Phase Monotonicity prevents mutation regardless. If the original was $\sigma \neq \mathsf{crystal}$, then neither value resides in a crystal region and Lemma 9.5 guarantees $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$. In either case, no mutable state is shared. □

Proof of Theorem 6.2. Suppose $\langle \rho, \mu, x \rangle \Downarrow \langle \mu, v'^{\sigma} \rangle$. By the operational semantics, only the E-Var rule applies. The rule specifies:

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

Thus $v' = \mathsf{deepclone}(v)$ where $\rho_{\text{raw}}(x) = v^{\sigma}$.

We must show: for any address $a \in \mathsf{mreach}(v'^{\sigma})$ and $b \in \mathsf{mreach}(v^{\sigma})$, we have $a \neq b$.

By Lemma 9.5, $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) \subseteq \mathsf{arena}(\mathcal{R})$.

If $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$, then value isolation holds trivially.

If there exists $a \in \mathsf{ptrs}(v') \cap \mathsf{ptrs}(v)$ with $a \in \mathsf{arena}(\mathcal{R})$, then by Lemma 9.6, a is excluded from mutable reachability for both values.

In all cases, $\mathsf{mreach}(v'^{\sigma}) \cap \mathsf{mreach}(v^{\sigma}) = \emptyset$. By Definition 9.4, the values share no mutable state. Furthermore, the store μ is unchanged by the E-Var rule, confirming that variable reads are pure observations with no side effects on memory. □

10 Proof of Theorem 6.3: Consuming Freeze

We restate the theorem for convenience and then provide a complete proof.

Theorem 6.3 (Consuming Freeze). *In strict mode ($\mathcal{M} = \mathsf{strict}$), after evaluating* $\mathbf{freeze}(x)$ *where x is an identifier, the variable x is no longer bound in the resulting environment:*

$$\langle \rho, \mu, \mathbf{freeze}(x) \rangle \Downarrow \langle \rho', \mu', v^{\mathsf{crystal}} \rangle \quad \Longrightarrow \quad x \notin \mathrm{dom}(\rho')$$

Proof. We structure the proof in four parts: (1) a direct derivation from the E-Freeze-Strict rule; (2) a demonstration that subsequent access to x produces an error; (3) a contrast with the casual-mode rule E-Freeze-Casual; and (4) a discussion of how the static phase checker cooperates with the runtime semantics.

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

Assume $\mathcal{M} = \mathsf{strict}$ and that we evaluate $\mathbf{freeze}(x)$ for some identifier x in configuration $\langle \rho, (\mathcal{F}, \mathcal{R}), \mathbf{freeze}(x) \rangle$. The only applicable evaluation rule is E-Freeze-Strict, whose premises and conclusion are:

E-Freeze-Strict

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

We verify each premise: (i) $\mathcal{M} = \mathsf{strict}$ holds by assumption; (ii) x is bound in ρ; (iii) $\mathsf{setphase}$ recursively sets the phase to $\mathsf{crystal}$; (iv) $\mathsf{freeze\_region}$ allocates a fresh region and deep-clones into it.

The resulting environment is $\rho' = \rho \setminus x$. By the definition of environment removal, $\mathrm{dom}(\rho \setminus x) = \mathrm{dom}(\rho) \setminus \{x\}$, so $x \notin \mathrm{dom}(\rho')$. □ (for the main claim)

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 is stuck, manifesting as a runtime error: $\mathsf{error}(\texttt{"undefined variable '}x\texttt{'"})$.

Part 3: Contrast with casual mode (E-Freeze-Casual).

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

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

Part 4: Cooperation of the static phase checker (PC-Freeze).

The static rule PC-Freeze ensures that in strict mode, **freeze** is never applied to an already-$\mathsf{crystal}$ value ($\sigma \neq \mathsf{crystal}$). This prevents wasteful double-freeze and cooperates 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*:

$$\mathbf{freeze} : \tau^{\mathsf{fluid}} \multimap \tau^{\mathsf{crystal}}$$

where $\multimap$ denotes 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 the safety benefits of linear types—*no fluid alias survives a freeze*—without global annotation burden. □

11 Proof of Forge Soundness

We restate the theorem and provide a complete proof.

Theorem 11.1 (Forge Soundness, Theorem 6.4 restated)**.** *The result of a* **forge** *block is always crystal-phased. That is, if evaluation of a forge block terminates with a value, then that value carries the* crystal *phase tag:*

$$\langle \rho, \mu, \textbf{forge } \{\ \bar{s}\ \} \rangle \Downarrow \langle \mu', v^{\sigma} \rangle \quad \Longrightarrow \quad \sigma = \mathsf{crystal}$$

Before proceeding to the main proof, we establish two supporting lemmas.

Lemma 11.2 (Setphase Totality)**.** *For any tagged value v^{σ} and target phase $\sigma' \in \mathsf{Phase}$, the operation $\mathsf{setphase}(v^{\sigma}, \sigma')$ is total and produces a value $v^{\sigma'}$ with σ' as its top-level phase tag. Moreover, every sub-value reachable from v also carries phase σ'.*

Proof. By structural induction on v.

Base cases. If v is a scalar (integer, float, boolean, string, or unit), then setphase sets $v.phase := \sigma'$ and returns. The result has phase σ'.

Inductive cases.

- *Array* $v = [v_1, \ldots, v_k]$. The operation sets $v.phase := \sigma'$ and recurses on each element v_i. By the induction hypothesis, each v_i obtains phase σ'.
- *Struct* $v = S\{f_1 : v_1, \ldots, f_k : v_k\}$. As above, the operation recurses on each field value.
- *Map* $v = \{s_1 : v_1, \ldots, s_k : v_k\}$. The operation recurses on each entry value.
- *Closure* $v = \langle \bar{x}, e, \rho \rangle$. The operation sets $v.phase := \sigma'$. (Closures are treated as opaque at the phase level; the captured environment ρ is not recursively re-phased.)

In all cases, the recursion terminates on the finite structure of v and sets σ' at every level, so the operation is total. □

Lemma 11.3 (Freeze-Region Phase Preservation)**.** *If v^{crystal} is a crystal-tagged value, then $\mathsf{freeze\_region}(v^{\mathsf{crystal}}, \mathcal{R}) = (v'^{\mathsf{crystal}}, \mathcal{R}')$ where v'^{crystal} has phase* crystal*.*

Proof. Deep cloning preserves the phase tag of the input. Setting the region identifier modifies only region metadata, not the phase tag. Therefore the output has phase crystal. □

Proof of Theorem 11.1. We proceed by case analysis on the result of evaluating the forge block's body.

Case 1: Normal completion. The block evaluates to w^{σ_w}. The rule E-Forge applies $\mathsf{setphase}(w^{\sigma_w}, \mathsf{crystal})$ (giving crystal by Lemma 11.2), then freeze_region (preserving crystal by Lemma 11.3). So $\sigma = \mathsf{crystal}$.

Case 2: Early return. A **return** signal carries a value w^{σ_w}. The forge block intercepts it and applies the same freeze protocol. By Lemmas 11.2 and 11.3, $\sigma = \mathsf{crystal}$.

Case 3: Error. The forge block propagates the error without producing a value. The theorem holds vacuously.

Case 4: Break or continue signals. These propagate unchanged; no value is produced. The theorem holds vacuously.

Since these four cases are exhaustive and in every case where a value is produced it has phase crystal, the theorem is proved. □

Proposition 11.4 (Static–Dynamic Agreement for Forge)**.** *The static phase-checking rule* PC-FORGE *infers* crystal *as the output phase of a forge expression, which agrees with the dynamic guarantee of Theorem 11.1.*

Proof. The rule PC-FORGE assigns phase crystal to the forge expression *unconditionally*—the inferred phases of the body statements do not influence the output phase. This mirrors the runtime behavior: the forge block's setphase step forces the result to crystal. The static checker's assertion can never be contradicted at runtime. □

Corollary 11.5 (Forge Compositionality)**.** *Forge blocks compose: nested forge blocks produce a crystal value at every level, and the outer forge block's freeze operation on an already-crystal value is idempotent with respect to the phase tag.*

Proof. By induction on the nesting depth n. The base case ($n = 1$) is Theorem 11.1. For the inductive step, regardless of the body's output phase σ_u, the outer forge applies setphase (forcing crystal) and freeze_region (preserving crystal). □

12 Proof of Heap Separation (Theorem 6.5)

We restate the theorem for convenience.

Theorem 6.5 (Heap Separation). *No pointer reachable from a crystal value with a valid region identifier $r \neq \bot$ appears in the fluid heap's allocation list:*

$$\forall\, v^{\mathsf{crystal}} \text{ with region } r.\ \forall\, a \in \mathsf{ptrs}(v).\ a \notin \mathrm{dom}(\mathcal{F})$$

We use the pointer set notation $\mathsf{ptrs}(v)$ from Definition 9.2, and establish auxiliary definitions and lemmas before giving the main proof.

12.1 Auxiliary Definitions

Definition 12.1 (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 12.2 (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 $a \in \mathrm{dom}(\mathcal{F})$.*

12.2 Key Lemmas

Lemma 12.3 (Arena–Fluid Disjointness)**.** *For every region R managed by $\mathcal{R}$ and every pointer $a \in \mathsf{arena}(R)$, we have $a \notin \mathrm{dom}(\mathcal{F})$.*

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 from fluid_alloc. □

Lemma 12.4 (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* arena(R)*.*

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

Lemma 12.5 (Deep Clone Allocation Coverage)**.** value_deep_clone(v) *allocates every pointer in* ptrs(v') *(where v' is the returned clone) through calls to* lat_alloc*,* lat_calloc*, or* lat_strdup*.*

Proof. By structural induction on the value type. Scalar types have empty pointer sets. For strings, arrays, structs, maps, and closures, every heap allocation in the clone is performed through the lat_$*$ family. Environment cloning dispatches to env_clone_arena when the arena is active (Lemma 12.6). □

Lemma 12.6 (Environment Arena Clone)**.** *When $g_{arena} \neq$* null*,* env_clone *dispatches to* env_clone_arena*, which allocates every pointer through* lat_alloc_routed*,* lat_calloc_routed*, and* lat_strdup_routed*—all of which delegate to the* lat_$*$ *family and hence route through* arena_alloc *by Lemma 12.4.*

Proof. By inspection of the env_clone implementation, which checks value_get_arena() and, when non-null, allocates all environment structures through the routed allocation functions. □

Lemma 12.7 (Region ID Completeness)**.** 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 12.8 (Value Free Removes Fluid Registrations)**.** *For a value v with v.region_id $= \perp$,* value_free(v) *calls* lat_free *on every pointer in* ptrs(v)*, removing them from $\mathcal{F}$. For a value with v.region_id $\neq \perp$,* value_free(v) *is a no-op.*

Proof. By inspection of value_free: the early return when *region_id* $\neq \perp$ prevents freeing arena-backed pointers. □

12.3 Main Proof

Proof of Theorem 6.5. We prove that after the freeze migration protocol completes, the resulting crystal value v' (with region ID r) satisfies ptrs$(v') \cap \mathrm{dom}(\mathcal{F}) = \emptyset$. The proof proceeds by induction on the seven steps of the protocol (Section 5).

Step 1: Create a fresh region. $R \leftarrow$ 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 12.4, all subsequent allocations go to arena(R). $\mathcal{F}$ unchanged.

Step 3: Deep-clone into R. $v' \leftarrow$ value_deep_clone(v_0) with $g_{arena} = R$. By Lemma 12.5, every pointer in ptrs(v') is allocated through lat_$*$. By Lemma 12.4, every such pointer is in arena(R). By Lemma 12.3, none appear in dom$(\mathcal{F})$. $\mathcal{F}$ unchanged.

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

Step 5: Set region ID recursively. $\mathsf{set\_region\_id\_recursive}(v', r)$. By Lemma 12.7, all sub-values carry region ID r, ensuring GC safety.

Step 6: Free the original. $\mathsf{value\_free}(v_0)$. By Lemma 12.8, 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:

$$\begin{aligned} &\mathsf{ptrs}(v') \subseteq \mathsf{arena}(R) && \text{(Step 3)} \\ &\mathsf{arena}(R) \cap \mathrm{dom}(\mathcal{F}) = \emptyset && \text{(Lemma 12.3)} \\ &\therefore\ \mathsf{ptrs}(v') \cap \mathrm{dom}(\mathcal{F}) = \emptyset \end{aligned}$$

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 prevents mutation of $\mathsf{ptrs}(v')$; (3) thaw produces independent copies (Theorem 6.6); (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 $\mathsf{assert\_crystal\_not\_fluid}$. □

13 Proof of Thaw Independence (Theorem 6.6)

We restate the theorem for convenience.

Theorem 13.1 (Thaw Independence, restated)**.** *The value produced by* $\mathbf{thaw}(e)$ *is a fresh, deep-cloned copy of the original with phase set to* fluid*. It shares no state with the source:*

$$\langle \rho, \mu, \mathbf{thaw}(e) \rangle \Downarrow \langle \mu', v'^{\mathsf{fluid}} \rangle \quad \Longrightarrow \quad \mathsf{ptrs}(v') \cap \mathsf{ptrs}([\![e]\!]) = \emptyset$$

In particular, thawing a crystal value does not invalidate the crystal region; the thawed copy resides in the fluid heap.

The proof uses the pointer set notation $\mathsf{ptrs}(v)$ from Definition 9.2.

13.1 Key Lemmas

Lemma 13.2 (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 9.5. 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 13.3 ($\mathsf{setphase}$ 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. □

13.2 Main Proof

Proof of Theorem 13.1. 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 value_thaw, which deep-clones v and sets the phase to fluid.

Since no freeze-to-region is in progress, g_arena = null. By Lemma 13.2, the clone v' has $\mathsf{ptrs}(v') \cap \mathsf{ptrs}(v) = \emptyset$ and all pointers reside in $\mathrm{dom}(\mathcal{F})$. By Lemma 13.3, the phase change does not touch the original. If the original was crystal in region R, R is unaffected. □ (Case 1)

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) env_get deep-clones the binding (Theorem 6.2 gives independence from the original); (2) value_thaw deep-clones again and sets phase to fluid; (3) a final value_deep_clone produces the return value; (4) env_set stores the thawed value.

By Lemma 13.2 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). □ (Case 2)

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

Corollary 13.4 (Crystal Safety Under Thaw)**.** *If* $\rho_{\mathrm{raw}}(x) = v^{\mathsf{crystal}}$ *with region* r*, then after evaluating* **thaw**(x)*:*

1. *The crystal region* $\mathcal{R}(r)$ *remains valid and unmodified.*
2. *Any other binding with* 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 13.2 and fluid heap residence. □