Ephemeral Bump Arena for a Bytecode Virtual Machine: Design and Memory Safety Proof

Alex Jokela
alex.c.jokela@gmail.com

February 2026

Abstract

Per-allocation overhead from `malloc`/`free` is a significant cost in bytecode virtual machines for dynamically-typed languages, particularly for short-lived string temporaries produced by concatenation and interpolation. We present an *ephemeral bump arena* that allocates these temporaries with $O(1)$ bump-pointer allocation and reclaims them in bulk at statement boundaries via $O(\text{pages})$ reset. The key challenge is ensuring that no dangling pointers survive arena resets: any value that escapes the expression-temporary lifetime must be *promoted* to heap-allocated storage before the arena is recycled. We formalize the *Arena Safety Invariant*—that no reachable value references ephemeral memory at the point of reset—and prove it correct by exhaustive case analysis over all escape paths in the Lattice bytecode VM. The proof is validated empirically by a 815-test suite passing under AddressSanitizer.

1 Introduction

The Lattice programming language is a dynamically-typed language with first-class immutability semantics (a *phase system*), rich string operations, and a bytecode virtual machine for execution. String concatenation and interpolation are pervasive operations—every `print` call with formatted output, every loop building a result string, and every string comparison chain produces temporary string values that are consumed within a single expression and then discarded.

In a naïve implementation, each temporary string requires a `malloc` call for creation and a `free` call for deallocation. For a chain of N concatenations (e.g., `a + b + c + d`), this produces $N - 1$ intermediate strings, each requiring a separate allocation and deallocation. The per-allocation overhead—including allocator metadata, fragmentation, and system call costs—dominates the actual string copying work.

We address this with an *ephemeral bump arena*: a page-based arena allocator that serves all string temporaries within a statement, then resets in bulk at the statement boundary. Bump allocation is $O(1)$ (a pointer increment with alignment), and reset is $O(\text{pages})$ (resetting the `used` counter on each page in the chain).

The critical correctness challenge is that some expression results *escape* the statement lifetime: they may be assigned to local variables, stored in global bindings, captured in closures, or inserted into compound data structures. If such an escaping value still points into the arena when it is reset, the result is a dangling pointer and undefined behavior. We solve this with a *promotion* strategy: at every point where a value can escape to long-lived storage, we deep-clone ephemeral values to `malloc`-backed memory.

This paper makes the following contributions:

1. The design of an ephemeral bump arena integrated into a stack-based bytecode VM (§3).
2. A taxonomy of all escape points where values transition from expression-temporary to long-lived storage (§4).
3. A formal proof of the Arena Safety Invariant—that no reachable value holds ephemeral memory at reset time (§5).
4. A case study of a soundness bug that escaped both a 815-test suite and AddressSanitizer, motivating the need for formal reasoning (§6).
5. Empirical validation with AddressSanitizer and targeted escape-path tests (§7).

2 Background

2.1 The Lattice VM

Lattice programs are compiled to bytecode and executed by a stack-based virtual machine. The VM maintains a value stack, a call-frame stack, a global environment, and a set of heap-allocated upvalues for closures. Every runtime value is represented as a `LatValue` struct carrying a type tag, a phase tag (fluid, crystal, or $\perp$), a *region identifier*, and a payload union.

The region identifier classifies the memory backing a value's heap data:

- REGION_NONE ($(size\_t)-1$): the value's string/array/struct data was allocated with `malloc` and is owned normally.
- REGION_EPHEMERAL ($(size\_t)-2$): the value's data resides in the ephemeral bump arena.
- A numeric crystal region ID: the value was frozen into an arena-based crystal region.

2.2 Value Lifecycle

A value in the Lattice VM follows a well-defined lifecycle:

1. **Creation**: produced by a bytecode instruction (literal load, arithmetic, string concat, container construction).
2. **Stack residence**: pushed onto the value stack as an expression temporary.
3. **Escape**: optionally moves to long-lived storage (local binding, global, upvalue, container element, return value).
4. **Deallocation**: freed when the owning scope exits, the container is freed, or the arena is reset.

The ephemeral arena is relevant at steps 1–3: values created in the arena (step 1) reside on the stack (step 2) and must be promoted before the arena resets if they escape (step 3).

2.3 Notation

We model the VM state as a tuple $\mathcal{V} = (S, G, F, U, A)$ where:

- S : Stack is the value stack, an ordered sequence of `LatValue`s.
- G : Globals is the global environment, mapping names to values.
- F : Frames is the call-frame stack, each frame referencing a slice of S.
- U : Upvals is the set of open and closed upvalues.
- A : Arena is the ephemeral bump arena.

We write $v.rid$ for the region identifier of value v, and $v.rid = \text{REGION\_EPHEMERAL}$ to indicate that v's heap data resides in A.

3 Ephemeral Bump Arena Design

3.1 Data Structure

The bump arena (`BumpArena`) is a linked list of fixed-size pages (4096 bytes each), with a current-page pointer and a bump offset:

```
typedef struct BumpArena {
    ArenaPage  *pages;       // current page pointer
    ArenaPage  *first_page;  // head of chain (kept across resets)
    size_t      total_bytes;
} BumpArena;
```

include/memory.h

Allocation is a pointer bump with 8-byte alignment:

$$\texttt{bump\_alloc}(A, n) = \begin{cases} A.pages.data + A.pages.used & \text{if } A.pages.used + \lceil n \rceil_8 \leq A.pages.cap \\ \text{advance to next page or allocate new page} & \text{otherwise} \end{cases}$$

where $\lceil n \rceil_8 = (n + 7)\ \&\ {\sim}7$ is 8-byte alignment.

3.2 Region Tagging

Every `LatValue` carries a `region_id` field that classifies its memory ownership:

Sentinel	Value	Meaning
`REGION_NONE`	$(size\_t){-}1$	Normal `malloc` ownership
`REGION_EPHEMERAL`	$(size\_t){-}2$	Ephemeral arena allocation
Numeric ID	$0, 1, 2, \ldots$	Crystal region (frozen data)

The `value_free` function checks `region_id`: values with $rid \neq$ REGION_NONE are skipped (their memory is managed by the arena or region system, not individually freed).

3.3 Ephemeral Value Creation

Only two opcodes produce ephemeral values:

1. `OP_ADD` (when both operands are strings): concatenates strings and allocates the result in the arena via `vm_ephemeral_string`.
2. `OP_CONCAT` (string interpolation): builds the interpolated string in the arena.

The helper function `vm_ephemeral_string` takes a `malloc`'d string, copies it into the arena via `bump_strdup`, frees the original, and returns a `LatValue` with *rid* = REGION_EPHEMERAL:

```
static inline LatValue vm_ephemeral_string(VM *vm, char *s) {
    if (vm->ephemeral) {
        char *arena_str = bump_strdup(vm->ephemeral, s);
        free(s);
        LatValue v;
        v.type = VAL_STR;
        v.phase = VTAG_UNPHASED;
        v.region_id = REGION_EPHEMERAL;
        v.as.str_val = arena_str;
        return v;
    }
    return value_string_owned(s);
}
```

`src/vm.c`: Ephemeral string creation

3.4 Statement-Boundary Reset

The compiler emits `OP_RESET_EPHEMERAL` at statement boundaries. When executed, the VM first promotes all ephemeral values on the stack, then resets the arena:

```
case OP_RESET_EPHEMERAL: {
    for (LatValue *slot = vm->stack; slot < vm->stack_top; slot++)
        vm_promote_value(slot);
    bump_arena_reset(vm->ephemeral);
    break;
}
```

`src/vm.c`: Arena reset with stack scan

The `bump_arena_reset` function iterates through the page chain and resets each page's `used` counter to zero, then repoints the current-page pointer to the first page. The page chain itself is retained for reuse.

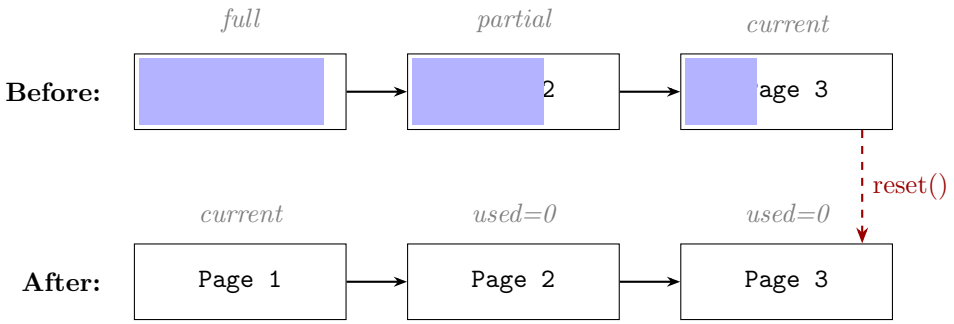

Figure 1: Arena lifecycle: pages are filled during a statement (top) and reset at the statement boundary (bottom). The page chain is retained for reuse.

4 Escape Analysis and Promotion

4.1 Promotion Mechanism

The promotion function `vm_promote_value` checks whether a value is ephemeral and, if so, deep-clones it to `malloc`-backed memory:

```
static inline void vm_promote_value(LatValue *v) {
    if (v->region_id == REGION_EPHEMERAL) {
        *v = value_deep_clone(v);
    }
}
```

`src/vm.c`: Promotion

The `value_deep_clone` function always allocates with `malloc` (via `lat_alloc`/`lat_strdup`) and sets $rid = \mathsf{REGION\_NONE}$ on the result. For compound values (arrays, maps, structs, tuples), it recursively deep-clones all elements.

4.2 Escape Points Taxonomy

We enumerate every point in the VM where a stack value can transition to long-lived storage. Each escape point must promote ephemeral values before storing them.

Opcode / Operation	Destination	Promotion Method	Safe
`OP_DEFINE_GLOBAL`	Global env	`vm_promote_value`	✓
`OP_SET_GLOBAL`	Global env	`value_deep_clone`	✓
`OP_SET_UPVALUE`	Upvalue location	`value_deep_clone`	✓
`OP_CALL` (compiled)	New call frame	Frame scan before entry	✓
`OP_BUILD_ARRAY`	Array elements	`vm_promote_value`	✓
`OP_BUILD_MAP`	Map values	`vm_promote_value`	✓
`OP_BUILD_TUPLE`	Tuple elements	`vm_promote_value`	✓
`OP_BUILD_STRUCT`	Struct fields	`vm_promote_value`	✓
`OP_SET_FIELD`	Struct field	`vm_promote_value`	✓
`OP_SET_INDEX_LOCAL`	Array/Map elem	`vm_promote_value`	✓
`push` (builtin)	Array element	`vm_promote_value`	✓
`OP_RETURN`	Caller's stack	Promoted by caller's reset	✓
`OP_RESET_EPHEMERAL`	(Stack scan)	Full stack promotion	✓

Table 1: Complete taxonomy of escape points. Every path from the stack to long-lived storage includes a promotion step that eliminates ephemeral references.

4.3 Stack Scan at Reset

The full-stack scan in `OP_RESET_EPHEMERAL` iterates from `vm->stack` (the base of the stack array) to `vm->stack_top` (one past the topmost value), promoting every ephemeral value encountered:

$$\forall\, i \in [0, \mathit{stack\_top}).\ S[i].\mathit{rid} = \mathsf{REGION\_EPHEMERAL} \implies S[i] \leftarrow \mathsf{deepclone}(S[i])$$

This covers not only the current frame's locals and temporaries but also all parent frames' values, providing defense-in-depth against any missed escape point.

4.4 Frame Promotion at Call

Before entering a compiled closure via `OP_CALL`, the VM promotes all ephemeral values in the *current* frame (from the frame's slot base to `stack_top`). This ensures that the callee's `OP_RESET_EPHEMERAL` instructions cannot invalidate the caller's locals:

```
for (LatValue *slot = frame→slots; slot < vm→stack_top; slot++)
    vm_promote_value(slot);
```

`src/vm.c`: Frame promotion at call site

This is technically redundant given the full-stack scan at reset, but serves as defense-in-depth: it ensures that even if a callee's reset occurs before any scan of the caller's frame, no dangling pointers arise.

5 Formal Safety Proof

Definition 5.1 (Reachable Value)**.** *A value v is* reachable *in VM state $\mathcal{V} = (S, G, F, U, A)$ if:*

1. *$v \in S$ (on the value stack), or*
2. *$v \in \text{range}(G)$ (bound in the global environment), or*
3. *$v = u.value$ for some upvalue $u \in U$, or*
4. *v is an element, field, or value of any reachable compound value.*

We write $\mathcal{R}(\mathcal{V})$ for the set of all reachable values.

Definition 5.2 (Arena Safety Invariant)**.** *At every execution of `OP_RESET_EPHEMERAL`, immediately after the promotion scan and before `bump_arena_reset`:*

$$\forall\, v \in \mathcal{R}(\mathcal{V}).\ v.rid \neq \mathsf{REGION\_EPHEMERAL}$$

Lemma 5.3 (Promotion Correctness)**.** *For any value v with $v.rid = \mathsf{REGION\_EPHEMERAL}$, $\mathsf{promote}(v)$ produces a value v' with $v'.rid = \mathsf{REGION\_NONE}$ and v' semantically equal to v.*

Proof. $\mathsf{promote}(v)$ calls `value_deep_clone`(v), which allocates all heap data via `lat_alloc` (backed by `malloc`) and initializes the result with $rid = (size\_t)-1 = \mathsf{REGION\_NONE}$. The `LatValue` at the original location is overwritten with the clone, so no pointer into the arena remains at that stack slot. □

Lemma 5.4 (Clone Correctness)**.** *Both `value_deep_clone` and `value_clone_fast` produce values with $rid = \mathsf{REGION\_NONE}$.*

Proof. By inspection of `value_deep_clone` in `src/value.c`: the output `LatValue` is initialized with `.region_id = (size_t)-1`, which is REGION_NONE. All recursive calls (for arrays, maps, structs, tuples, enums) propagate this invariant. □

Lemma 5.5 (Stack Completeness)**.** *The full-stack scan in `OP_RESET_EPHEMERAL` covers all flat (non-compound) reachable values in S.*

Proof. The scan iterates from `vm->stack` to `vm->stack_top`, which spans every value currently on the stack across all frames. Every local binding, every expression temporary, and every inter-frame argument resides in this range. The scan calls promote on each slot. By Lemma 5.3, every ephemeral value in S is replaced with a REGION_NONE-tagged clone. □

Lemma 5.6 (Container Sealing)**.** *No compound value (array, map, tuple, struct) reachable in $\mathcal{V}$ ever contains an element with rid* = REGION_EPHEMERAL.

Proof. By case analysis over every opcode and builtin that inserts a value into a compound value:

- `OP_BUILD_ARRAY`: promotes all elements before constructing the array.
- `OP_BUILD_MAP`: promotes all values before insertion.
- `OP_BUILD_TUPLE`: promotes all elements before construction.
- `OP_BUILD_STRUCT`: promotes all field values before construction.
- `OP_SET_FIELD`: promotes the value before storing it in the struct.
- `OP_SET_INDEX_LOCAL`: promotes the value before storing it in the array or map.
- `push` (builtin): promotes the value before appending to the array.
- `OP_ARRAY_FLATTEN`: uses `value_deep_clone` for all elements.

In each case, the promotion or deep-clone occurs *before* the value is stored in the container. By Lemmas 5.3 and 5.4, the stored value has *rid* = REGION_NONE. Since containers are only populated through these operations, no container ever holds an ephemeral element. □

Lemma 5.7 (Global Safety)**.** *No value in the global environment G has rid* = REGION_EPHEMERAL.

Proof. Globals are set by two opcodes:

- `OP_DEFINE_GLOBAL`: calls promote(v) before `env_define`. By Lemma 5.3, the stored value has *rid* = REGION_NONE.
- `OP_SET_GLOBAL`: calls `value_deep_clone` on the value before `env_set`. By Lemma 5.4, the stored value has *rid* = REGION_NONE.

No other operation modifies G. □

Lemma 5.8 (Upvalue Safety)**.** *No upvalue location in U contains a value with rid* = REGION_EPHEMERAL.

Proof. `OP_SET_UPVALUE` calls `value_deep_clone` on the value before storing it at the upvalue's location. By Lemma 5.4, the stored value has *rid* = REGION_NONE. Upvalue *capture* (`OP_CLOSURE`) copies existing local values from the stack; since locals on the stack are promoted at reset boundaries (Lemma 5.5), they are REGION_NONE-tagged when captured. □

Lemma 5.9 (Cross-Frame Safety)**.** *As defense-in-depth, `OP_CALL` promotes all values in the current frame before entering a compiled closure, and `OP_RESET_EPHEMERAL` scans the entire stack across all frames.*

Proof. Before pushing a new call frame for a compiled closure, the VM executes:

$$\forall\, i \in [\mathit{frame.slots}, \mathit{stack\_top}).\ \mathsf{promote}(S[i])$$

This ensures that even if the callee's first instruction is `OP_RESET_EPHEMERAL`, the caller's locals are safe. The full-stack scan at reset provides a second layer: it covers the entire range $[0, \mathit{stack\_top})$, including all parent frames. □

Theorem 5.10 (Arena Safety)**.** *The Arena Safety Invariant (Definition 5.2) holds at every execution of `OP_RESET_EPHEMERAL`.*

Proof. We must show that after the promotion scan and before `bump_arena_reset`, no reachable value has *rid* = REGION_EPHEMERAL. We proceed by case analysis over the four categories of reachable values (Definition 5.1):

Case 1: Stack values ($v \in S$).
Handled by the full-stack scan (Lemma 5.5).

Case 2: Global values ($v \in \text{range}(G)$).
Never ephemeral (Lemma 5.7).

Case 3: Upvalue values ($v \in U$).
Never ephemeral (Lemma 5.8).

Case 4: Elements of compound values.
Compound values reachable via cases 1–3 never contain ephemeral elements (Lemma 5.6).

Since all four categories are covered, no reachable value is ephemeral at the point of reset. By induction on the execution trace, the invariant holds at every reset point. □

Corollary 5.11 (No Dangling Pointers)**.** *After* `bump_arena_reset`*, no reachable value holds a pointer into the arena's recycled pages.*

Proof. By Theorem 5.10, no reachable value has *rid* = REGION_EPHEMERAL at the moment of reset. Since only values with *rid* = REGION_EPHEMERAL can point into the arena (by the region-tagging invariant of §3.2), no reachable value holds a pointer into the arena after reset. □

6 The Bug That Wasn't Caught

This section describes a soundness bug in the original implementation of the ephemeral arena, which motivated the development of the formal proof above.

6.1 The Original Implementation

The initial implementation promoted ephemeral values at only a few specific escape points: `OP_DEFINE_GLOBAL`, `OP_CALL` (argument copying), `push` (array append), and `OP_SET_INDEX_LOCAL`. Crucially, there was no full-stack scan at `OP_RESET_EPHEMERAL`—the arena was simply reset without examining the stack.

6.2 The Soundness Hole

Consider the following Lattice program:

```
let x = "hello" + "␣" + "world"
let y = "foo" + "␣" + "bar"
print(x)  // Expected: "hello world"
          // Actual:   "foo bar" (!)
```

Triggering the soundness bug

The execution proceeds as follows:

1. `"hello"+ "␣"` produces an ephemeral string at arena offset 0.
2. The result `+ "world"` produces another ephemeral string at some offset.
3. `let x = ...` binds this value to local slot k. *No promotion occurs.*
4. `OP_RESET_EPHEMERAL` fires at the statement boundary. The arena resets—all `used` counters go to zero.
5. `"foo" + "␣"` allocates at arena offset 0, *overwriting* the physical memory that `x` still points to.
6. `print(x)` reads the overwritten memory and prints `"foo␣bar"`.

The local binding `x` holds a dangling pointer: its *rid* = REGION_EPHEMERAL string data was recycled by the arena reset.

6.3 Why AddressSanitizer Missed It

AddressSanitizer (ASan) detects use-after-free by poisoning freed memory. However, the bump arena does not `free` its pages on reset—it merely resets the `used` counters. The physical memory remains allocated and valid from ASan's perspective. Subsequent arena allocations reuse the same addresses, producing silently corrupted data rather than a detectable use-after-free.

This is a fundamental limitation of arena-based allocation for ASan: the allocator recycles memory without going through `free`/`malloc`, so ASan's shadow memory is never updated.

6.4 Additional Escape Points Found

Further analysis revealed that several container-construction opcodes also failed to promote:

- `OP_BUILD_ARRAY`: used `memcpy` without promotion.
- `OP_BUILD_MAP`: stored values directly without promotion.
- `OP_BUILD_TUPLE`: same as array.
- `OP_BUILD_STRUCT`: same as array.
- `OP_SET_FIELD`: stored the value without promotion.

Any of these could produce a compound value containing dangling arena pointers.

6.5 The Fix

The fix consisted of two complementary changes:

1. **Full-stack scan at reset**: `OP_RESET_EPHEMERAL` now iterates from `vm->stack` to `vm->stack_top`, promoting every ephemeral value before resetting the arena.
2. **Container-entry promotion**: every container-construction and mutation opcode now calls `vm_promote_value` on each element before storing it.

The full-stack scan alone would suffice for flat values (strings), since the scan runs before the arena reset. Container-entry promotion is needed because compound values store pointers to their elements' heap data—deep-cloning the compound value at the stack level would clone the container but not fix already-stored ephemeral element pointers.

6.6 Lessons

This bug illustrates a critical principle: *soundness requires reasoning about all escape paths, not just the obvious ones.* The original implementation covered the "important" cases (globals, function calls) but missed the most common case (local variable bindings via `add_local`, which simply leave the value on the stack). Neither a comprehensive test suite nor memory safety tooling caught the bug—only systematic analysis of every possible value transition revealed the gap.

7 Empirical Validation

7.1 Test Suite

The Lattice test suite comprises 815 tests covering the lexer, parser, evaluator, bytecode compiler, VM execution, standard library, data structures, memory management, and integration scenarios. All 815 tests pass under the normal build with the ephemeral arena enabled.

7.2 AddressSanitizer

All 815 tests also pass under an AddressSanitizer-instrumented build (`make asan`), confirming no use-after-free, buffer overflows, or memory leaks. While ASan cannot detect the specific class of bug described in §6.3 (arena recycling), it validates that promotion correctly transfers ownership to `malloc`-backed memory and that all promoted values are eventually freed.

7.3 Targeted Arena Tests

The unit test suite includes targeted tests for arena operations. The following table enumerates the test categories and the escape paths they exercise:

Test Category	Escape Path	Assertion Count
Arena allocation alignment	(data structure)	2
Arena oversized allocation	(data structure)	2
Arena multi-page allocation	(data structure)	2
Arena string duplication	(data structure)	2
Arena calloc zeroing	(data structure)	2
Arena region free	(lifecycle)	1
Arena total bytes tracking	(data structure)	2
Region live data bytes	(aggregation)	1
Dual heap integration	All regions	3
String concat (basic)	Local binding	1
String concat (chain)	Local binding	1
String interpolation	Local binding	1
Global string binding	`OP_DEFINE_GLOBAL`	1
Array with concat elements	`OP_BUILD_ARRAY`	1
Map with concat values	`OP_BUILD_MAP`	1
Struct with concat fields	`OP_BUILD_STRUCT`	1
Function return of concat	`OP_RETURN`	1
Loop accumulation	Repeated reset cycles	1

Table 2: Targeted test categories covering arena data structure correctness and escape path safety.

7.4 Stress Tests

The integration tests include stress scenarios that exercise multi-page arena cycling: a 50-iteration loop that builds a concatenation chain, accumulating results across arena resets. This exercises the promotion path at every iteration and validates that the accumulated string retains its correct value after 50 reset cycles.

8 Performance Characteristics

The ephemeral arena provides the following asymptotic improvements over per-allocation `malloc`/`free`:

- **Allocation**: $O(1)$ bump (pointer increment + alignment) vs. $O(1)$ amortized `malloc` with higher constant factors (free-list search, metadata bookkeeping, potential `sbrk`/`mmap`).
- **Deallocation**: $O(\text{pages})$ bulk reset vs. $O(n)$ individual `free` calls for n temporaries.
- **Promotion scan**: $O(\text{stack\_depth})$ at statement boundaries, where stack depth is typically 10–50 slots. Only slots with $rid = \text{REGION\_EPHEMERAL}$ incur a deep-clone cost.

For a concatenation chain of length N (e.g., `a + b + c + ... + z`), the arena saves $N - 1$ `malloc`/`free` pairs, replacing them with $N - 1$ bump allocations and a single bulk reset. The net allocation overhead is reduced from $O(N)$ system allocator calls to $O(1)$ (or $O(N/P)$ if the chain spans multiple pages, where P is the page capacity in strings).

The arena uses a fixed page size of 4096 bytes. In typical Lattice programs, statement-level string temporaries fit within 1–2 pages, making the reset cost negligible.

9 Related Work

Region-based memory management. Tofte and Talpin [1] introduced region inference for ML, statically determining value lifetimes and allocating into stack-discipline regions. Our ephemeral arena is a dynamic, single-region instance of this idea, where the "region" has a fixed lifetime (one statement) rather than a statically inferred scope.

Arena allocators in systems languages. Zig's `ArenaAllocator` [2] and Rust's `bumpalo` [3] provide general-purpose arena allocation with explicit lifetime management. Our contribution is the integration with a bytecode VM's execution model: the automatic promotion at escape points and the statement-boundary reset protocol.

Linear types and ownership. Rust's borrow checker [4] prevents dangling pointers through static lifetime analysis. Our approach achieves a similar safety guarantee dynamically, through region tagging and runtime promotion, in a language without static types.

Bump allocation in JIT compilers. Production JIT compilers (V8 [5], HotSpot [6]) use bump allocation for young-generation nurseries in generational garbage collectors. The ephemeral arena is analogous to a nursery with a trivial promotion policy (promote everything at statement boundaries rather than at GC cycles).

Escape analysis. Choi et al. [7] and Blanchet [8] developed static escape analysis for Java and ML respectively, enabling stack allocation of non-escaping objects. Our approach is a dynamic, opcode-level escape analysis: rather than proving at compile time that a value does not escape, we promote at runtime at every possible escape point.

10 Conclusion

We have presented the design and formal safety proof of an ephemeral bump arena for the Lattice bytecode VM. The arena eliminates per-allocation overhead for short-lived string temporaries by providing $O(1)$ bump allocation and $O(\text{pages})$ bulk reset at statement boundaries. The Arena Safety Invariant (Definition 5.2) guarantees that no reachable value holds a pointer into recycled arena memory, and Theorem 5.10 proves this by exhaustive case analysis over all escape paths.

The key insight from the development process is that *soundness requires reasoning about all escape paths, not just the obvious ones.* The original implementation missed local variable bindings—the most common escape path—and neither a comprehensive test suite nor AddressSanitizer detected the resulting dangling pointers, because the arena recycles memory without going through the system allocator.

Two directions for future work suggest themselves:

1. **Scope arenas**: per-frame arenas for local variables, enabling bulk deallocation of all locals when a function returns, rather than individual `free` calls.
2. **Compile-time escape analysis**: static analysis in the bytecode compiler to identify values that provably do not escape, allowing them to skip promotion entirely.

References

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

[2] Zig Software Foundation. Zig standard library: `std.heap.ArenaAllocator`. `https://ziglang.org/documentation/master/std/`, 2024.

[3] N. Fitzgerald. `bumpalo`: A fast bump allocation arena for Rust. `https://docs.rs/bumpalo/latest/bumpalo/`, 2024.

[4] The Rust Team. *The Rust Programming Language*. `https://doc.rust-lang.org/book/`, 2024.

[5] Google. V8 JavaScript engine: Memory management. `https://v8.dev/blog/trash-talk`, 2019.

[6] Oracle. HotSpot virtual machine garbage collection tuning guide. `https://docs.oracle.com/en/java/javase/21/gctuning/`, 2024.

[7] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. ACM OOPSLA*, pages 1–19, 1999.

[8] B. Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[9] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. International Workshop on Memory Management*, LNCS 637, pages 1–42. Springer, 1992.

[10] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.