Compiler-Transparent Ownership: A Multidimensional Modifier Algebra for Safe, Fast, and Injection-Free Systems Programming

Anonymous Author(s)

Abstract

We present U, a systems programming language built around a **multidimensional modifier algebra**: three orthogonal binary axes—ownership ($\pm R$), mutability ($\pm M$), nullability ($\pm N$)—plus five hardware domain modifiers (`+R` heap, `+G` GPU, `+C` cache, `+A` async-surviving). Each annotation is simultaneously a *safety guarantee* and an *exact compiler oracle*: $-R$ replaces escape analysis; $-M$ replaces alias analysis; $-N$ replaces null-flow analysis; `+A` replaces liveness analysis for suspension frames. The result is a **dual pit of success**—the design space where the easy path is simultaneously the correct path for two distinct audiences. For *programmers*: the zero-annotation default ($-R\ -M\ -N$) delivers memory safety, race freedom, null safety, and async correctness without borrow-checker annotations, `Option` unwrapping, or async/await coloring. For *compilers*: the same zero-annotation path eliminates the entire escape, alias, points-to, and liveness analysis pipeline—replaced by direct reads of type annotations. No existing language achieves both simultaneously.

We identify and close three open problems. **(1)** *The function-coloring problem* [19]: async annotations infect call chains in every async/await language (Rust, JavaScript, Python), and goroutines (Go) solve coloring but not frame size. Our **Red-Blue Freedom Theorem** and **Fiber Frame Minimization Theorem** prove that U's stackful fibers with the `+A` domain solve both simultaneously. **(2)** *The injection safety problem*: SQL injection and XSS arise from untyped string interpolation; library solutions are bypassable. Our **Template Safety Theorem** proves that `html`/`sql` interpolations are escaped at compile time—injection is a type error, structurally impossible. **(3)** *The i18n correctness problem*: missing translation keys and placeholder mismatches are runtime errors in every existing framework. Our **Localization Completeness Theorem** proves these are compile-time type errors in U.

We prove sixteen theorems in total. The **Modifier-Optimization Correspondence Theorem**—the central result—establishes that each annotation in the multidimensional algebra is a sound compiler transformation license; safety results (ARC-Freedom, Race-Freedom, Proxy Safety, Type Soundness) follow as corollaries. The **Signature Completeness Theorem** establishes that function signatures are machine-verified contracts covering all modifier dimensions simultaneously. We compare U against Rust, Go, JavaScript, Python, and C++, showing U occupies a previously unoccupied intersection: Go's concurrency ergonomics, Rust's safety, C's performance, and compile-time injection and i18n safety that no existing language provides.

CCS Concepts: • **Software and its engineering** → *Functional languages*; **Concurrent programming structures**; *Multiparadigm languages.*

Keywords: ownership types, type systems, concurrency, memory safety, local-first, reference counting, proxy abstraction

1 Introduction

1.1 The Dual Pit of Success

Every type annotation in U serves two clients simultaneously. The programmer reading a signature learns what invariants hold. The compiler reading the same signature learns which analysis to skip. This double-duty is not coincidental—it is the central design principle of the language, which we formalize as a **multidimensional modifier algebra** with the property of *compiler-transparent ownership*: type annotations are compiler oracles. The algebra has three binary modifier dimensions ($\pm R$, $\pm M$, $\pm N$) and five hardware domains, each dimension orthogonal and independently composable. The zero-annotation default is the pit of success for both audiences: cheapest for the compiler, safest for the programmer, requiring no annotations in the common case.

The phrase "pit of success" [1] describes a system designed so the easy path is the correct path. Most programming language research targets one audience: programmers (safety, ergonomics) or compilers (optimization, analysis). U targets both simultaneously, through the same mechanism.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

POPL '27, January 2027, Mexico City, Mexico

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

The programmer's pit. A programmer writing a function in U with no type annotations gets, by default, local immutable parameters ($-R\ -M$) and local mutable objects ($+M\ -R$). These defaults are simultaneously the safest (no dangling references, no data races) and the fastest (no heap allocation, no ARC, no synchronization). Sharing is opt-in via $+R$; concurrency policy is declared, not scattered. The programmer cannot accidentally write an unsafe or slow program without explicitly opting in to the constructs that introduce cost and risk.

The compiler's pit. A compiler processing U code with no optimizations enabled still produces optimal code for the common case—because the type annotations are a complete oracle for every major transformation:

- *Escape analysis* is replaced by the $-R$ annotation. The compiler need not analyze whether a value escapes its scope; the type says it does not.
- *Alias analysis* for mutation is replaced by $-M$. An immutable reference cannot alias a mutable write.
- *Points-to analysis* for shared objects is replaced by $+R\,+M$. The exact points-to set is the proxy; no flow analysis needed.
- *ARC elimination* follows immediately from $-R$: no heap location exists, no refcount was created, nothing to eliminate.
- *Synchronization elimination* follows from $+R\ -M$: immutable shared data needs no locks.

In typical languages, these analyses are among the most expensive in a compiler, are approximate, and interact with each other in complex ways. In U, they are exact, free, and independent—because the programmer has already provided the answer in the type. We call this property *compiler-transparent ownership*.

1.2 Three Open Problems, One Mechanism

Two problems have resisted clean solutions across all major systems languages. We state them precisely so the contributions can be evaluated against them.

The function-coloring problem [19]. In any language with explicit async/await (JavaScript, Python, Rust, C#), async functions have a different type from sync functions. Calling an async function from a sync function requires making the sync function async; the annotation propagates upward through the call graph, splitting every codebase into two incompatible worlds. Go solves this with goroutines (no coloring) but pays in frame size—goroutines save the entire call stack at suspension. Rust solves the frame-size problem with stackless coroutines (lean frames) but reintroduces coloring—`async fn` and `fn` are distinct types. *No existing language solves both simultaneously with a formal guarantee.* U does: the Red-Blue Freedom Theorem (Theorem 13.2) proves that all functions may call all functions without annotation change; the Fiber Frame Minimization Theorem (Theorem 13.7) proves that the suspension frame contains exactly the `+A`-annotated live values—no more.

The injection safety problem. SQL injection and cross-site scripting (XSS) are among the most pervasive security vulnerabilities in production software. Both arise from the same root cause: string interpolation into a structured context (SQL, HTML) without grammar-aware escaping. Rust's `sqlx` and `askama` address this as libraries; Go's `html/template` provides partial protection; TypeScript has no enforcement. All library approaches can be bypassed—the protection is advisory, not structural. *No existing language makes injection attacks type errors.* U does: the Template Safety Theorem (Theorem 18.3) proves that every `{expr}` interpolation in a `html`, `sql`, or `msg` template is escaped or parameterized at compile time; raw interpolation requires the explicit `{{expr}}` unsafe marker.

A *third problem* is worth naming: *internationalization correctness*. Every major web application needs locale-aware strings, but no language verifies at compile time that translation keys exist, that placeholder counts match, or that format types are consistent. The result is runtime exceptions in production when a translator omits a key or introduces a placeholder mismatch. The Localization Completeness Theorem (Theorem 21.6) makes these errors type errors, closing the problem with the same mechanism as template safety.

All three problems share a common structure: they arise from the absence of type-level information the programmer already possesses ("this value survives suspension"; "this string enters a SQL context"; "this key must be in the locale bundle"). U makes all three explicit in the type system and proves the corresponding safety properties from the types.

1.3 The Modifier System

U's type system rests on two orthogonal binary modifiers applied to every reference:

- $\pm R$ (*ownership*): $-R$ means locally owned, non-shared, stack-resident; $+R$ means shared, retained, heap-resident, potentially remote.
- $\pm M$ (*mutability*): $-M$ means immutable through this reference; $+M$ means mutable.

The eight-combination modifier space and its cost model are summarized in Table 1. Each quadrant corresponds to an exact set of compiler licenses (Table 1).

1.4 A New Programming Experience

The combination of solved problems enables a programming experience that does not currently exist:

> *High-performance systems code where async and sync are the same thing, null is safe by default, injection and missing-translation errors are impossible by construction, and the compiler needs*

Mod.	Compiler license	Analysis replaced	Cost
$-R$	Stack-allocate; eliminate ARC	Escape analysis	Zero heap, zero ARC
$-R\ -M$	Duplicate freely; no copy	Alias analysis	Zero-copy pass
$-R$ (callee)	Pass by ref, no copy	Interprocedural escape	Same-frame pass
$+R\ -M$	No synchronization	Immutability analysis	Lock-free reads
$+R+M$	Exact points-to = proxy	Points-to analysis	Proxy-bounded
$-R$ closure	Stack-allocate closure	Closure escape	Zero-alloc lambda

Table 1. Each modifier annotation is an exact optimization license. The compiler receives these facts for free, without analysis.

no analysis to produce optimal code—because the programmer's type annotations are the analysis.

Concretely: a U web server handles thousands of concurrent requests with no explicit async annotation on any handler (Red-Blue Freedom); never null-dereferences a request field without a compile-time check ($-N$ default); never produces an XSS response regardless of user input (Template Safety); allocates closures in tight loops on the stack with no heap traffic ($-R$ default); and compiles with no escape analysis, alias analysis, or async-propagation analysis—because each modifier annotation has already answered the relevant question.

No existing language delivers all of these simultaneously. Go comes closest on concurrency ergonomics but lacks null safety, injection safety, and the optimization oracle. Rust comes closest on safety and performance but has the coloring problem, requires `Option` for nullability, and relies on library-level injection protection. U occupies the intersection.

1.5 Safety as a Corollary

The safety results—ARC-Freedom, Race-Freedom, Proxy Safety—are not the primary goal of the type system. They are *corollaries* of the optimization correspondence. When the compiler is licensed to eliminate all ARC operations for $-R$ terms (because none exist), it follows immediately that no ARC overhead is incurred. When the compiler knows all $+R+M$ mutations pass through a proxy, race-freedom follows because the proxy serializes accesses.

This inversion—safety as a consequence of optimization transparency, rather than optimization as a consequence of safety analysis—is the conceptual contribution of U's type system. It means the same formal results serve two constituencies simultaneously.

Contributions.

1. **Compiler-transparent ownership and the Modifier-Optimization Correspondence Theorem** (§5–6): a core calculus λ_U with three modifier dimensions ($\pm R$, $\pm M$, $\pm N$) and a five-element modifier system, plus a compilation scheme and proof that each annotation is a sound transformation license. Safety results (ARC-Freedom, Race-Freedom, Proxy Safety, Type Soundness) follow as corollaries.
2. **Red-Blue Freedom and Fiber Frame Minimization** (§13): a stackful fiber model with the `+A` domain, solving Nystrom's function-coloring problem and minimizing suspension-frame overhead to exactly the `+A`-live values.
3. **Signature Completeness Theorem** (§9): function signatures are machine-verified contracts covering all three modifier dimensions; callers can trust signatures without reading implementations.
4. **Template Safety Theorem** (§18): context-aware templates (`html`, `sql`, `msg`) make injection attacks structurally impossible as compile-time type errors.
5. **Systematic language comparisons** (§22): formal comparison against Rust, Go, JavaScript, Python, and C++, identifying the design point U occupies.

2 The U Language: Design Overview

We present the aspects of U's design that interact directly with the formal system. Surface syntax details (unified branching, generator-based iteration, tab-only indentation) are in a companion technical report; here we focus on the modifier system and the constructs formalized in λ_U.

2.1 Notation Correspondence

The paper uses two notations for modifiers. The *formal notation* ($+R/-R$, $+M/-M$, $+N/-N$) is used in the core calculus λ_U, type rules, and theorem statements. The *surface notation* uses `+X` domain annotations on type declarations. Table 2 gives the correspondence.

2.2 Design Rationale for Modifiers

The three-modifier system emerged through iteration. An early design used `+L`/`-L` for locality, but locality is more naturally the *absence* of retention: $-R$ (not retained, not remote, not refcounted) aligns directly with runtime semantics. An early `+C` (const) was replaced by $-M$ (not mutable): mutability is a capability, and capability-oriented design favors opt-in over opt-out. Both dimensions default to negation-of-capability ($-R$, $-M$), so the zero-annotation path is simultaneously cheapest and safest.

Formal	Surface	Meaning
$-R$ (default)	(no annotation)	local, stack-resident
$+R$	`+R`	remote, heap, ARC-managed
$+M$	`+M`	mutable (often implicit with `+R`)
$-N$ (default)	(no annotation)	guaranteed non-null
$+N$	`+N` suffix	nullable / may be `None`
`+G`	GPU domain	GPU device memory
`+C`	Cache domain	near-memory, prefetch-friendly
`+A`	Async domain	survives fiber suspension

Table 2. Modifier notation: formal (λ_U calculus) and surface U (U language) notation. The zero-annotation path corresponds to the $-R\ -M\ -N$ formal default.

Kw.	Meaning	Notes
`a`	async / suspension	fiber yield point
`b`	break	inside `w` generators
`c`	continue	inside `w` generators
`d`	class definition	no struct/class split
`e`	error / throw	propagates up fiber
`f`	named function	`=>` for lambdas
`r`	return target	`r => value`
`t`	this / self	
`w`	generator / loop	warns if no `b` reachable
`y`	yield	inside `w`
`None`	null value	only multi-letter keyword

Table 3. Complete U keyword set. All keywords are one letter except `None`. Multi-letter requirement for identifiers prevents collisions.

2.3 The Modifier System

Table 3 gives the complete keyword set.

Modifiers appear as + or − prefixed to single letters placed after the type annotation, or directly after the function keyword `f`.

Example 2.1 (Modifier annotations).

```
f distance(point: Point, origin: Point)
    distance = sqrt(point.x*point.x + point.y*point.y)

// Local mutable (class default): no annotation needed
counter = Counter()
counter.value = counter.value + 1

// Shared immutable: explicit +R, safe to propagate freely
config: Config +R -M = loadConfig()

// Shared mutable: explicit +R +M, proxy-mediated
cache: Cache +R +M = Cache()
cache.put(key, value)  // dispatches through proxy
```

Example 2.2 (Escaping closures). *Local closures capture by reference at zero cost. Closures that escape their scope must be marked +R; the compiler promotes captured locals:*

```
// Local closure: stack-allocated, zero cost
transform = value => value * 2
result = [1,2,3].x(transform)

// Shared/escaping closure: heap-allocated, refcounted
handler = f +R(event) = processEvent(event)
button.onClick(handler)   // handler outlives this scop
```

2.4 Copy-on-Write, Classes, and Iteration

Shallow copy via `.c()` is the only mechanism for producing an independent $-R$ value from a $+R$ object, making promotion costs explicit. All data types are classes; the modifier system determines layout and cost rather than a `struct`/`class` distinction. A class with all $-R$ instances is stack-allocated; a class with $+R$ instances is heap-allocated and proxy-capable. Iteration uses generators in place of `for` loops, unifying loops, async streams, and event sources under one abstraction. These design choices are orthogonal to the formal results; the λ_U calculus abstracts over them.

3 U in Practice: Idiomatic Examples

Before the formal treatment, we present idiomatic U code to give the reader a feel for the language. These examples are representative programs, not toy constructs. Notice that the zero-annotation path— local, immutable, non-null—requires no modifier annotations at all; they appear only where the programmer deliberately opts into sharing, mutability, nullability, or hardware placement.

Example 1: Data Pipeline

```
// Compute the top-5 active users from a log file
// No annotations needed: everything is local, immutabl

f topUsers(logPath: S): [UserSummary]
    lines = readLines(logPath)                  // [S]
    // local, inferred
    parsed = lines
        .x(line => parseLogEntry(line))         // parse eac
        .x(entry => entry ?? skip)              // drop fail
        .x(entry => (                           // transform
            userId:  entry.userId,
            count:   entry.actionCount,
            lastSeen: entry.timestamp
        ))
    sorted = parsed.sortBy(s => s.count)        // sort desc
    r => sorted.x((s, index) =>
        index < 5 & s                           // take firs
    )
```

Every closure is stack-allocated (−*R* default); no heap traffic in the pipeline body. The ?? on line 5 discards failed parses without an explicit null check. The type of `lines` is inferred as `[S] -R -M -N`.

Example 2: Safe HTTP Handler

```
// HTTP handler -- note: no 'a' annotation on the caller.
// The fiber scheduler handles suspension transparently.

d Server
    routes: [Route] +R -M    // shared immutable route table
    db:     Database +R +M   // shared mutable, proxy-mediated

f handleRequest(server: Server, request: Request): Response +R
    route = server.routes.find(r => r.matches(request.path))
    route ?? r => None                  // 404: no matching route

    user = a server.db.getUser(request.userId)
// suspends fiber
    user ?? r => Response.unauthorized()
// no user found

    body = html`
        <h1>{route.title}</h1>
        <p>Welcome, {user.displayName}!</p>
        <ul>{user.recentItems.x(item =>
            html`<li>{item.name}</li>`
        )}</ul>
    `
    r => Response.ok(body)
```

The `a` suspension on line 9 yields the current fiber; the caller (`main`, a non-`a` function) is not affected. The HTML template auto-escapes all `{expr}` interpolations; XSS is structurally impossible. The ?? pattern on lines 7 and 11 returns early on failure without nested conditionals.

Example 3: Concurrent Counter with Policy

```
// A shared counter that multiple fibers increment safely.
// +R +M triggers proxy mediation; policy choice is at declaration.

d Counter [policy: MVCC]
    value: I

    f increment(counter: Counter +R +M)
        counter.value = counter.value + 1   // routed through MVCC proxy

    f get(counter: Counter +R -M): I
        r => counter.value                  // snapshot read, no lock

// Usage: ten fibers incrementing concurrently
shared: Counter +R +M = Counter()
(0..10).x(index =>
    a shared.increment()    // each increment is fiber-safe
)
result = shared.get()       // deterministic: exactly 1
```

`shared` is +*R* +*M*: the MVCC proxy serializes writes while allowing concurrent snapshot reads. The caller needs no explicit lock, mutex, or channel. The data-race freedom guarantee (Theorem 7.3) holds by the type, not by analysis.

Example 4: Class, Self Reference, and Error Handling

```
// A simple stack. Methods use 't' for self.
// The 'e' keyword throws an error up the fiber.
d Stack
    items: [Any] +R +M

    f push(stack: Stack +R +M, item: Any)
        t.items.append(item)

    f pop(stack: Stack +R +M): Any
        t.items.length == 0 & e StackError("pop from em
        r => t.items.removeLast()

    f isEmpty(stack: Stack): B
        r => t.items.length == 0

// Usage
stack: Stack +R +M = Stack()
stack.push(42)
stack.push("hello")
top = stack.pop()                       // "hello"
empty = stack.isEmpty()                 // false
```

`t` refers to the receiver within a method body. The `e` on line 9 is a guarded throw: the `&` short-circuits so the error fires only when the stack is empty. Errors propagate up the fiber's call stack without `Result<T,E>` wrapping at every call site.

Example 5: Locale-Aware UI

```
// Internationalized greeting page.
// Compile-time verification: all keys must exist in th

// Locale bundle (e.g. en.json):
// { "greeting": "Hello, {name}!",
//   "itemCount": "You have {count} item(s).",
//   "noItems": "Your cart is empty." }

f greetingPage(user: User, cart: Cart +N): S
    items = cart?.items ?? []           // safe: cart m
    cartMsg = items.length > 0
        & msg`{itemCount, count: items.length}`
        | msg`{noItems}`
    r => msg`{greeting, name: user.displayName}` ++ car
```

If the locale bundle is missing `greeting`, `itemCount`, or `noItems`, the program does not compile. The `?.` on line 9 propagates `None` safely; the `??` falls back to an empty array. The `&/|` conditional on lines 10–11 reads naturally as: "if items, show count; otherwise show empty message."

4 The Core Calculus λ_U

We formalize a core calculus λ_U capturing the essential mechanisms: local and shared references, objects with fields, function abstraction and application, shallow copy, and proxy dispatch. We omit generators, iterators, and width-specialized types, which are orthogonal to the ownership results.

4.1 Syntax

Definition 4.1 (Modifiers and Types)**.**

$$\begin{array}{rrcl} \textit{Ownership mod.} & \rho & ::= & +R \mid -R \\ \textit{Mutability mod.} & \mu & ::= & +M \mid -M \\ \textit{Base types} & \beta & ::= & \mathtt{I} \mid \mathtt{N} \mid \mathtt{D} \mid \mathtt{S} \mid \mathtt{B} \mid \mathtt{T} \\ \textit{Object types} & \tau & ::= & \beta \mid \{\overline{f:\tau}\}^{\rho,\mu} \mid (\overline{\tau}) \xrightarrow{\rho} \tau \\ \textit{Proxy types} & \pi & ::= & \mathsf{Proxy}[\tau^{+R,+M}] \end{array}$$

An object type $\{f_1{:}\tau_1, \ldots, f_n{:}\tau_n\}^{\rho,\mu}$ carries its ownership and mutability modifiers as part of the type. Function types $(\overline{\tau}) \xrightarrow{\rho} \tau$ carry an ownership modifier on the closure itself; $\rho = -R$ means the closure does not escape its enclosing scope.

Definition 4.2 (Terms)**.**

$$\begin{array}{rcl} e & ::= & x \mid c \mid \{\overline{f = e}\} \mid e.f \mid e.f := e' \mid \\ & & \mathbf{f}^{\rho}(\overline{x{:}\tau}).\, e \mid e\,(\overline{e}) \mid \mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ e' \mid \\ & & e.\mathsf{c}() \mid \mathsf{proxy}(e) \mid e\,[\mathsf{dispatch}]\,(\overline{e}) \end{array}$$

where x ranges over variable names, c over base-type constants, ρ over ownership modifiers, and τ over types.

The term $e\,[\mathsf{dispatch}]\,(\overline{e})$ is *proxy dispatch*: it applies a mutation through the proxy wrapping e. Surface U notation writes this as ordinary field assignment or method call on a $+R\,+M$ object; the elaborator inserts the [dispatch] annotation.

4.2 Values and Heaps

Definition 4.3 (Values and Heap)**.** *Values are:*

$$v \ ::= \ c \mid \ell \mid \mathbf{f}^{\rho}(\overline{x{:}\tau}).\, e$$

where ℓ ranges over locations*. We distinguish two kinds of location:*

- Stack locations ℓ^{-R}*: bound to the current stack frame, never stored in a $+R$ context.*
- Heap locations ℓ^{+R}*: managed by ARC; carry a reference count $n \geq 1$ and an optional proxy wrapper.*

A heap H maps heap locations to heap objects $\langle\, \overline{f = v},\ n,\ \mathsf{proxy?}\,\rangle$ *where n is the reference count and* proxy? *is a flag indicating whether this object is proxy-wrapped.*

$$\frac{v \text{ is a value}}{\langle \sigma, H,\ \mathbf{let}\ x{:}\tau^{-R,\mu} = v\ \mathbf{in}\ e\rangle \longrightarrow \langle \sigma[\ell^{-R} \mapsto v], H,\ e\{\ell^{-R}/x\}\rangle}\ [\textsc{Let-Local}]$$

$$\frac{v \text{ is a value} \quad \ell^{+R} \notin \mathrm{dom}(H)}{\langle \sigma, H,\ \mathbf{let}\ x{:}\tau^{+R,\mu} = v\ \mathbf{in}\ e\rangle \longrightarrow \langle \sigma,\ H[\ell^{+R} \mapsto \langle v, 1, \mathrm{false}\rangle],\ e\{\ell^{+R}/x\}\rangle}\ [\textsc{Let-Shar}$$

$$\frac{H(\ell^{+R}) = \langle \overline{f = v},\ n,\ \mathrm{false}\rangle \quad n > 1}{\langle \sigma, H,\ \ell^{+R}.\mathsf{c}()\rangle \longrightarrow \langle \sigma[\ell^{-R} \mapsto \overline{f = v}],\ H,\ \ell^{-R}\rangle}\ [\textsc{Copy}]$$

$$\frac{H(\ell^{+R}) = \langle \overline{f = v},\ n,\ \mathrm{true}\rangle \quad P \text{ policy-ok}}{\langle \sigma, H,\ \ell^{+R}\,[\mathsf{dispatch}]\,(\overline{v})\rangle \longrightarrow \langle \sigma,\ H[\ell^{+R} \mapsto \langle P(\overline{f = v}, \overline{v}),\ n,\ \mathrm{true}\rangle],\ \ell^{+R}\rangle}\ [\textsc{Proxy-}$$

Figure 1. Selected λ_U reduction rules. Let-Local allocates to the stack (no heap entry); Let-Shared allocates to the heap with refcount 1; Copy produces a stack copy of a shared object; Proxy-Dispatch mutates a proxy-wrapped heap object through policy P.

A stack σ maps stack locations to stack records $\overline{f = v}$ *(reference-count-free).*

4.3 Operational Semantics

We give a small-step semantics over configurations $\langle \sigma, H, e\rangle$. We write $e\{v/x\}$ for capture-avoiding substitution. Selected rules are given in Figure 1; the full set is in the appendix.

Note that Let-Local places the value in the stack σ only; no entry in H is created, and no reference count is allocated. This is the operational foundation of the ARC-Freedom theorem.

4.4 Concurrent Operational Semantics

To state Race-Freedom precisely, we extend λ_U with a parallel composition operator and an interleaved reduction relation.

Definition 4.4 (Parallel Programs and Thread Pools)**.** *A* thread pool *is a finite multiset $\mathcal{T} = \{e_1, \ldots, e_n\}$ of expressions. A* concurrent configuration *is a triple $\langle \sigma, H, \mathcal{T}\rangle$.*

Definition 4.5 (Concurrent Reduction)**.** *The concurrent reduction relation $\langle \sigma, H, \mathcal{T}\rangle \xrightarrow{c} \langle \sigma', H', \mathcal{T}'\rangle$ is defined by two rules:*

$$\frac{\langle \sigma, H, e_i\rangle \longrightarrow \langle \sigma', H', e_i'\rangle \quad e_i \in \mathcal{T}}{\langle \sigma, H, \mathcal{T}\rangle \xrightarrow{c} \langle \sigma', H', (\mathcal{T} \setminus \{e_i\}) \cup \{e_i'\}\rangle}\ [\textit{Thread-Step}]$$

$$\frac{e_i, e_j \in \mathcal{T} \quad i \neq j \quad H(\ell) \text{ is modified by } e_i \text{ and accessed by } e_j}{\langle \sigma, H, \mathcal{T}\rangle \text{ has a data race on } \ell}\ [\textit{Race}]$$

A data race on ℓ *occurs when two distinct threads can in the same step both access ℓ with at least one write. We say a concurrent program is* race-free *if no reachable concurrent configuration has a data race.*

The key axiom for the proxy model is that Proxy-Dispatch steps are *non-interleaved*: two concurrent Proxy-Dispatch steps on the same heap location ℓ^{+R} are sequentialized by the

proxy's internal locking (enforced by the policy invariant, Definition 7.4). This is stated as:

Axiom 4.6 (Proxy Non-Interleaving). *No concurrent configuration contains two simultaneous* Proxy-Dispatch *reductions on the same heap location. The proxy implementation guarantees mutual exclusion at the policy boundary.*

This axiom is an abstract requirement on proxy implementations; concrete policies (MVCC, Actor, Mutex) each satisfy it by construction, as shown in Table 5.

5 Compiler-Transparent Ownership

This section states and proves the central result: the Modifier-Optimization Correspondence Theorem. We define compiler transformations as program transformations on λ_U terms, and show each transformation is semantics-preserving exactly when the corresponding modifier condition holds.

5.1 A Compilation Scheme

To ground the optimization claims concretely, we define a compilation function $[\![\cdot]\!]$ from λ_U terms to a typed assembly language TAL_U with explicit stack and heap regions.

Definition 5.1 (Target Language TAL_U and Compilation). TAL_U *has two allocation primitives:* $\mathsf{salloc}(v)$ *(stack-allocate* v*, freed on frame pop) and* $\mathsf{halloc}(v)$ *(heap-allocate* v*, managed by ARC). The compilation function* $[\![e]\!]^{\mu}_{\rho}$ *is parameterized by the modifier of the term being compiled:*

$$[\![\mathbf{let}\ x{:}\tau^{-R,\mu} = v\ \mathbf{in}\ e]\!] \;=\; x \leftarrow \mathsf{salloc}(v);\ [\![e]\!]$$

$$[\![\mathbf{let}\ x{:}\tau^{+R,\mu} = v\ \mathbf{in}\ e]\!] \;=\; x \leftarrow \mathsf{halloc}(v);\ \mathsf{retain}(x);\ [\![e]\!];\ \mathsf{release}(x)$$

$$[\![\mathbf{f}^{-R}(\overline{x{:}\tau}).\, e]\!] \;=\; \mathsf{closure\_salloc}([\![e]\!], \overline{x})$$

$$[\![\mathbf{f}^{+R}(\overline{x{:}\tau}).\, e]\!] \;=\; \mathsf{closure\_halloc}([\![e]\!], \overline{x})$$

All other constructs compile homomorphically.

Theorem 5.2 (Compilation Correctness). *The compilation function* $[\![\cdot]\!]$ *is correct: for all well-typed closed* e *with* $\emptyset \vdash e : \tau$, $[\![e]\!]$ *terminates with the same observable value as* e*, using* salloc *exclusively for* $-R$*-typed sub-terms and* halloc *exclusively for* $+R$*-typed sub-terms.*

Proof sketch. By structural induction on e. The modifier annotation on each **let** binding directly determines which allocation primitive the compilation function selects. Correctness of observable values follows from the semantics-preservation of the compilation scheme (the compiled code produces the same values as the source semantics). The salloc/halloc split follows immediately from the syntactic structure of the compilation function. □ □

Corollary 5.3 (Optimization Oracle). *For any well-typed program, the compiler can determine the allocation strategy, ARC placement, and synchronization requirements for every subexpression by a single pass over the type annotations, without any dataflow analysis.*

Definition 5.4 (Semantics-Preserving Transformation). *A program transformation* T *is* semantics-preserving *for expression* e *under environment* Γ *if for all heaps* H *and stacks* σ*:* $\langle \sigma, H, e\rangle \longrightarrow^* v$ *iff* $\langle \sigma', H', T(e)\rangle \longrightarrow^* v$ *for some* σ', H' *that agree with* σ, H *on all observable locations, and* v *is the same observable value.*

Theorem 5.5 (Modifier-Optimization Correspondence). *Let* $[\![\cdot]\!]$ *be the compilation scheme of Definition 5.1 (Section 5.1). The following hold for all well-typed* λ_U *programs:*

1. ***Stack Allocation.*** *If* $\Gamma \vdash e : \tau^{-R,\mu}$*, then* $[\![e]\!]$ *contains only* salloc *allocations for* e*'s bindings—no* halloc *call is emitted.*
2. ***ARC Elimination.*** *If* $\Gamma \vdash e : \tau^{-R,\mu}$*, then* $[\![e]\!]$ *contains no* retain *or* release *operations for* e*'s bindings.*
3. ***Same-Frame Passing.*** *If* $\Gamma \vdash e : \tau^{-R,\mu}$ *is passed to* $\mathbf{f}^{-R}(x{:}\tau^{-R,\mu}).\, e'$*, then* $[\![e]\!]$ *passes the stack address of* e *directly, without emitting a copy instruction.*
4. ***Synchronization Elimination.*** *If* $\Gamma \vdash e : \tau^{+R,-M}$*, then* $[\![e]\!]$ *emits no memory fence, lock acquire, or atomic load instruction for reads of* e*.*
5. ***Exact Points-To.*** *If* $\Gamma \vdash e : \tau^{+R,+M}$*, then the points-to set of* e *in the compiled output is the singleton* $\{[\![\mathsf{proxy}(e)]\!]\}$*; no alias analysis is required to establish this.*
6. ***Closure Stack Allocation.*** *If* $\Gamma \vdash \mathbf{f}^{-R}(\overline{x{:}\tau}).\, e' : (\overline{\tau}) \xrightarrow{-R} \tau_r$*, then* $[\![\mathbf{f}^{-R}\cdots]\!]$ *emits* salloc *for the closure record, not* halloc*.*

Each statement is a direct consequence of the compilation scheme structure and the corresponding typing rule; in each case, the compiler decision is a syntactic read *of the modifier annotation, not an analysis result.*

Proof. We prove each part.

(1) Stack Allocation. By the ARC-Freedom theorem (Theorem 7.1), no evaluation of $e : \tau^{-R,\mu}$ invokes Let-Shared. Therefore the allocation behavior of e is entirely captured by Let-Local (stack allocation). Replacing the implementation of Let-Local to use a stack frame rather than any heap structure is semantics-preserving because the operational semantics of Let-Local depends only on the binding $\sigma[\ell^{-R} \mapsto v]$, not on the physical allocation site.

(2) ARC Elimination. By the ARC-Freedom theorem, no $+R$ locations are created during evaluation of $e : \tau^{-R,\mu}$. No $+R$ location means no reference count exists; eliminating retain/release operations on a non-existent count is trivially semantics-preserving.

(3) Same-Frame Passing. By the Stack-Frame Confinement lemma (Lemma 6.4), the callee cannot retain e beyond the call. Therefore passing a reference to the stack location of e is equivalent to passing a copy: the callee reads the same value in both cases, and the caller's location remains valid throughout the call. Semantics-preservation follows.

(4) Synchronization Elimination. An object of type $\tau^{+R,-M}$ is immutable by the $-M$ modifier. The only applicable

rule is Field-Read; no write rule types against $-M$ objects. Concurrent Field-Read steps commute (reads of the same immutable value produce the same result in any order), so no synchronization is required for correctness.

(5) Exact Points-To. By the Proxy Safety theorem (Theorem 7.5), every mutation of $e : \tau^{+R,+M}$ passes through $\text{proxy}(e)$. No rule modifies the object denoted by e except via Proxy-Dispatch applied to $\text{proxy}(e)$. The points-to set—the set of memory locations that may be modified by a mutation operation on e—is therefore exactly $\{\text{proxy}(e)\}$. This is a constant the compiler can compute at type-check time; no flow-sensitive analysis is needed.

(6) Closure Stack Allocation. A closure of type $(\overline{\tau}) \xrightarrow{-R} \tau_r$ carries modifier $-R$, meaning by part (1) it can be stack-allocated. Its captured variables are $-R$ typed (by the sharing discipline; a $-R$ closure cannot capture $+R$ variables without promotion). Therefore the entire closure record— function pointer plus captured environment—resides on the stack and requires no heap allocation. □ □

Corollary 5.6 (Zero-Analysis Optimization)**.** *A compiler for U can perform stack allocation, ARC elimination, copy elimination, synchronization elimination, and exact alias resolution* without running any dataflow analysis. *Each optimization fires on a direct read of the type annotation. The type checker and the optimizer share the same pass.*

Remark 5.7 (Approximate vs. Exact)**.** *In a language without modifier annotations, a compiler must* approximate *these properties. Escape analysis may conservatively heap-allocate a value that could have stayed on the stack. Alias analysis may conservatively synchronize accesses that are actually race-free. Points-to analysis produces sets that are over-approximations. In U, each corresponding fact is* exact*: $-R$ means exactly "not on the heap," not "probably not on the heap." The type system eliminates the entire class of precision-loss that drives most compiler research in this area.*

6 Type System

6.1 Typing Rules

A typing environment Γ maps variables to types. The judgment $\Gamma \vdash e : \tau$ means e has type τ in environment Γ. Selected rules appear in Figure 2; the full set is in the appendix.

6.2 Subtyping

The modifier dimensions induce a natural subtyping relation. Intuitively, a more permissive reference type is a subtype of a more restrictive one: having mutable access implies having read access; having local access implies you can treat a value as non-shared.

Definition 6.1 (Modifier Subtyping)**.** *The subtyping relation $<:$ on modifier pairs is the least relation satisfying:*

$$\frac{}{\tau^{\rho,+M} <: \tau^{\rho,-M}}\ [\textsc{Mut-Sub}] \qquad \frac{}{\tau^{-R,\mu} <: \tau^{+R,\mu}}\ [\textsc{Own-Sub}]^{\dagger}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash e_i : \tau_i \text{ for all } i}{\Gamma \vdash \{\overline{f_i = e_i}\} : \{\overline{f_i{:}\tau_i}\}^{-R,+M}}\ [\textsc{Obj-Local}]$$

$$\frac{\Gamma, \overline{x{:}\tau} \vdash e : \tau_r \quad \rho \in \{-R, +R\}}{\Gamma \vdash \mathbf{f}^{\rho}(\overline{x{:}\tau}).e : (\overline{\tau}) \xrightarrow{\rho} \tau_r}\ [\textsc{Fun}]$$

$$\frac{\Gamma \vdash e_0 : (\overline{\tau}) \xrightarrow{\rho} \tau_r \quad \Gamma \vdash e_i : \tau_i \text{ for all } i}{\Gamma \vdash e_0(\overline{e_i}) : \tau_r}\ [\textsc{App}]$$

$$\frac{\Gamma \vdash e : \tau^{+R,\mu}}{\Gamma \vdash e.\text{c}() : \tau^{-R,+M}}\ [\textsc{Copy}]$$

$$\frac{\Gamma \vdash e : \tau^{+R,+M}}{\Gamma \vdash \text{proxy}(e) : \text{Proxy}[\tau^{+R,+M}]}\ [\textsc{Proxy-Wrap}]$$

$$\frac{\Gamma \vdash e : \{\overline{f{:}\tau}\}^{+R,+M} \quad \Gamma \vdash e_j : \tau_j}{\Gamma \vdash e\,[\text{dispatch}]\,(e_j) : \{\overline{f{:}\tau}\}^{+R,+M}}\ [\textsc{Proxy-Dispatch}]$$

Figure 2. Selected λ_U typing rules. Object literals default to $-R+M$ (Obj-Local). Field mutation on a $+R+M$ object requires proxy dispatch (Proxy-Dispatch). A $-R$ object cannot directly become $+R$; it must be heap-allocated via Let-Shared, which is elaborated from explicit $+R$ type annotations.

where † marks a rule that is not *included.*

Ownership subtyping in the † direction is deliberately excluded: a $-R$ (local) value may not *be treated as $+R$ (shared) without explicit heap promotion via Let-Shared. This exclusion is what makes the sharing discipline enforceable—if $-R <: +R$ held, a local value could silently become shared, defeating ARC-Freedom.*

Only mutability subtyping holds: $\tau^{\rho,+M} <: \tau^{\rho,-M}$ (a mutable reference can be used where an immutable one is expected, but not vice versa). This is the standard covariance of immutability.

Remark 6.2 (Why Not $-R <: +R$)**.** *Most ownership type systems allow a unique/local reference to be "forgotten" into a shared one (upcast). U deliberately prevents this implicit upcast. The programmer must write an explicit annotation (+R type ascription, triggering Let-Shared) to promote a local value to a shared one. The cost of this promotion—a heap allocation and refcount initialization—is made visible at the point of promotion rather than hidden in a subtyping coercion.*

6.3 Modifier Inference and Defaults

In surface U, most modifier annotations are omitted. The elaborator applies the following defaults before type-checking:

1. Function parameters without annotations receive $-R-M-N$ (local, immutable, non-null).
2. Object literal types receive $-R+M-N$ (local, mutable, non-null).
3. A closure without an explicit $+R$ receives $-R$.

4. A variable declared with an explicit $+R$ type undergoes heap allocation via Let-Shared elaboration.
5. A value annotated $+N$ may hold `null`; field access or method dispatch on a $+N$ value is a type error without a prior null-check dispatch (?? coalescing or & guard).

The triple default $-R\ -\ M\ -\ N$ is the *pit of success in all three dimensions*: the cheapest execution path, the safest ownership, and null-safety by default. All three costs (heap allocation, mutation, null-dereference) are opt-in.

These defaults mean that programs which do not share state—the common case—require no annotations and type-check with full safety.

6.4 Well-Typedness and the Sharing Discipline

The key *sharing discipline* enforced by the type system is:

Definition 6.3 (Sharing Discipline)**.** *A program satisfies the sharing discipline if:*

1. *No $-R$ value is stored in a $+R$ context without an intervening Let-Shared elaboration.*
2. *No mutation of a $+R+M$ object occurs except through proxy dispatch.*
3. *No $+R$ closure captures $-R$ variables directly (the compiler promotes them via copying).*
4. *No field access or method call is performed on a $+N$ value without an intervening null check (the Null-Check typing rule).*

The type rules enforce the sharing discipline by construction: Copy is the only way to move a $+R$ value to $-R$, and Proxy-Dispatch is the only way to mutate a $+R+M$ object.

6.5 Stack-Frame Safety for $-R$ Passing

A subtle property the sharing discipline must establish is that passing a $-R$ value to a callee is safe—i.e., the callee cannot retain the value beyond the caller's stack frame.

Lemma 6.4 (Stack-Frame Confinement)**.** *If $\Gamma \vdash e : \tau^{-R,\mu}$ and e is passed as an argument to a function $\mathbf{f}^{-R}(x{:}\tau^{-R,\mu}).\ e'$, then no evaluation of e' stores a reference to the argument in any $+R$ context.*

Proof. The function carries modifier $-R$, meaning the closure itself does not escape. By the sharing discipline (Def. 6.3), no $-R$ value can be stored in a $+R$ context without an intervening Let-Shared elaboration, which requires an explicit $+R$ annotation. Since x has type $\tau^{-R,\mu}$ and the function body e' is type-checked under $\Gamma, x{:}\tau^{-R,\mu}$, any attempt to store x into a $+R$-typed location requires a type ascription that changes the modifier to $+R$— which the sharing discipline prohibits without an explicit Let-Shared. Therefore the argument cannot be retained beyond the call. □

This lemma is the formal counterpart of the informal "same stack" insight: local values passed to callees remain local because the type system prevents the callee from promoting them to the heap without an explicit $+R$ annotation visible at the call site.

Remark 6.5 (Scope of Lemma 6.4)**.** *The lemma assumes the function body e' contains no explicit $+R$ ascription on x. This is a property of the* declared function type*, not of e''s internal structure: if the function type is $(\tau^{-R,\mu}) \xrightarrow{-R} \tau_r$, then any use of x inside e' that would require a $+R$ ascription is a type error at the call site. Formally, the type-checker rejects programs where a $-R$-typed parameter appears in a $+R$ binding position without explicit promotion. The sharing discipline (Def. 6.3, clause 1) enforces this.*

7 Safety as Corollaries

The safety results follow as corollaries of the Modifier-Optimization Correspondence Theorem (Theorem 5.5). We restate them in safety terms for clarity; full proofs are in the appendix.

7.1 ARC-Freedom

Theorem 7.1 (ARC-Freedom (Corollary of Theorem 5.5.2))**.** *Let e be a closed, well-typed λ_U expression with $\emptyset \vdash e : \tau^{-R,\mu}$. No evaluation sequence $\langle \sigma_0, H_0, e\rangle \longrightarrow^* \langle \sigma_k, H_k, v\rangle$ allocates any heap location or performs any atomic increment or decrement operation.*

Proof sketch. By induction on typing derivations and evaluation steps.

No heap allocation: The only rule that allocates a heap location is Let-Shared, which applies only when the declared type carries $+R$. Since $e : \tau^{-R,\mu}$, no subexpression of e has a $+R$ type under $\Gamma = \emptyset$ (by the sharing discipline; a $-R$ expression cannot contain a $+R$ subexpression without an explicit annotation introducing it). Therefore Let-Shared is never invoked.

No atomic operations: Atomic increment and decrement occur only when a heap location's reference count changes. Since no heap locations are allocated, no reference counts exist and no atomic operations occur.

Inductive preservation: The sharing discipline is preserved across all reduction steps (proved as a corollary of the Preservation lemma below). Therefore the invariant holds throughout evaluation. □ □

Corollary 7.2 (Local Execution is Zero-Cost)**.** *A program fragment that operates entirely within the $-R$ quadrant—the default for function bodies whose parameters are not annotated—executes with no ARC overhead, no heap pressure, and no atomic contention, regardless of call depth.*

7.2 Race-Freedom

We model concurrency as a standard interleaved thread semantics. A *data race* is a configuration in which two threads concurrently access the same memory location with at least one write.

Theorem 7.3 (Race-Freedom (Corollary of Theorem 5.5.4–5))**.** *In any well-typed λ_U program, no execution contains a data race.*

Proof sketch. Using the concurrent operational semantics (Definition 4.5), we show no reachable concurrent configuration satisfies the Race condition. We case-split exhaustively on memory location type.

Case $-R$ (stack locations): Stack locations ℓ^{-R} are created by Let-Local and stored in thread-private stacks σ_i. By the sharing discipline (Def. 6.3), no $-R$ value is stored in any $+R$ heap context; therefore no $-R$ location appears in H. Since threads share only H, no two threads hold a reference to the same ℓ^{-R}. The Race condition requires two threads to access the same ℓ; impossible for $-R$.

Case $+R\,-M$ (shared immutable): Heap locations of type $\tau^{+R,-M}$ admit only Field-Read steps (no write rule applies to $-M$ objects, by the typing rules). The Race condition requires at least one write; impossible.

Case $+R\,+M$ (shared mutable): All modifications of ℓ^{+R} carrying the proxy flag must go through Proxy-Dispatch. By Axiom 4.6 (Proxy Non-Interleaving), no two concurrent Proxy-Dispatch steps on the same ℓ^{+R} occur simultaneously. Two concurrent threads accessing ℓ^{+R} are therefore serialized at the proxy; at most one thread modifies ℓ^{+R} at any step, satisfying the Race-free condition. □ □

7.3 Proxy Safety

Definition 7.4 (Policy-Conforming Proxy)**.** *A proxy wrapping $\tau^{+R,+M}$ is* policy-conforming *with respect to invariant $\mathcal{I}$ if, for every dispatch call with pre-state s satisfying $\mathcal{I}$ and arguments $\overline{v}$, the post-state $P(s,\overline{v})$ satisfies $\mathcal{I}$.*

Theorem 7.5 (Proxy Safety (Corollary of Theorem 5.5.5))**.** *Let e be well-typed with $\emptyset \vdash e : \tau^{+R,+M}$. Every mutation of the object denoted by e during evaluation is mediated by a Proxy-Dispatch step. If the proxy wrapping e is policy-conforming with respect to invariant $\mathcal{I}$ and the initial state satisfies $\mathcal{I}$, then $\mathcal{I}$ holds throughout all reachable states.*

Proof sketch. The only reduction rule that modifies a heap location marked $+R\,+M$ is Proxy-Dispatch. The type rule Proxy-Dispatch is the only typing rule that types a mutation expression on a $+R\,+M$ object. By the Preservation lemma, well-typedness is maintained across reductions, so direct field mutation ($e.f := e'$) on a $+R\,+M$ object is never typeable and hence never occurs. The invariant preservation follows by induction on evaluation steps: each step either does not touch the heap object (so $\mathcal{I}$ is vacuously preserved) or applies P, which preserves $\mathcal{I}$ by policy-conformance. □ □

Corollary 7.6 (Policy Plug-in)**.** *Any policy P satisfying policy-conformance gives a safe concurrency model for $+R\,+M$ objects. In particular, MVCC, actor-mailbox serialization, CRDT merge, and two-phase-locking policies are all admissible.*

7.4 Type Soundness

We prove soundness via the standard Progress and Preservation lemmas.

Lemma 7.7 (Preservation)**.** *If $\Gamma \vdash e : \tau$ and $\langle \sigma, H, e\rangle \longrightarrow \langle \sigma', H', e'\rangle$, then $\Gamma \vdash e' : \tau$.*

Proof sketch. By case analysis on the reduction rule applied. For Let-Local and Let-Shared, the new binding has exactly the declared type. For Copy, the rule produces $\tau^{-R,+M}$ from $\tau^{+R,\mu}$, matching the type of Copy in Figure 2. For Proxy-Dispatch, the result type is $\tau^{+R,+M}$, matching the typing rule. Substitution preserves typing by the standard substitution lemma (proved by induction on type derivations). □ □

Lemma 7.8 (Progress)**.** *If e is a closed, well-typed expression, then either e is a value or there exist σ', H', e' such that $\langle \sigma, H, e\rangle \longrightarrow \langle \sigma', H', e'\rangle$.*

Proof sketch. By induction on the typing derivation. All base cases (variables, constants, values) are immediate. For compound expressions, the inductive hypotheses supply reductions for subexpressions; canonical forms lemmas (standard; in appendix) ensure that values of each type admit the required reduction. The key case is Proxy-Dispatch: a $+R\,+M$ object is always heap-allocated (by Let-Shared) and therefore always carries the proxy flag, so Proxy-Dispatch always applies. □ □

Theorem 7.9 (Type Soundness)**.** *If $\emptyset \vdash e : \tau$, then evaluation of e does not get stuck: either it produces a value of type τ or it diverges.*

Proof. Immediate from Lemmas 7.7 and 7.8 by standard subject-reduction argument. □ □

8 The Proxy Model

The proxy abstraction is the mechanism by which $+R\,+M$ objects achieve safe concurrent mutation without borrow-checker restrictions. We give a brief formal treatment.

8.1 Three-Tier Runtime Separation

The multidimensional modifier system induces a three-tier runtime architecture that most languages leave implicit. Table 4 summarizes the tiers; the key property is that each tier is *induced by type annotation*, not by runtime dispatch or compiler heuristic.

The transition from Shared to Distributed requires no source-code changes: the proxy implementation is swapped for a remote-dispatch variant. Because the type system already mandates that all $+R\,+M$ mutations pass through a proxy (Theorem 7.5), a distributed deployment is a policy decision, not a language change.

Tier	Storage	ARC	Sync	Reach
Local ($-R$)	stack	none	none	same frame
Shared ($+R$)	heap	atomic	proxy	cross-thread
Distributed ($+R$ remote)	remote	protocol	policy	cross-machine

Table 4. Three-tier runtime separation induced by the $\pm R$ modifier. The compiler statically determines which tier each object inhabits; no runtime tier detection is needed.

8.2 The Proxy as Universal Runtime Boundary

The proxy abstraction in U is intentionally more general than a concurrency primitive. A single $+R+M$ reference may, depending on the deployed policy, represent: a heap-allocated object with MVCC semantics; an actor mailbox serializing messages; a CRDT-backed replicated value; a remote object accessed via RPC; a transactional workspace with snapshot isolation; or a collaboratively edited document with operational transformation. All of these share the same surface type and the same type-level guarantee (Proxy Safety, Theorem 7.5). The language does not need separate primitives for each concurrency model; the proxy is the boundary, and the policy is what varies.

This positions U's $+R+M$ objects closer to *distributed systems abstractions*—similar in spirit to Erlang processes or Akka actors—than to ordinary shared-memory concurrency primitives, while retaining the static safety of a type system.

8.3 Proxy Policies

Definition 8.1 (Proxy Policy)**.** *A proxy policy P for object type $\tau^{+R,+M}$ is a function:*

$$P : \mathsf{State}(\tau) \times \mathsf{Mutation}(\tau) \to \mathsf{State}(\tau) \times \mathsf{Response}$$

where $\mathsf{State}(\tau)$ *is the set of field-value maps for* τ*,* $\mathsf{Mutation}(\tau)$ *is the set of mutation requests (field assignments, method calls), and* Response *is the set of observable results. P is* safe *if for all policies-conforming states and mutations, the result is also a policy-conforming state.*

The surface language allows communities of objects to declare their policy as a class annotation:

```
class Counter +R +M [policy: MVCC]
    value: I
```

The compiler selects the proxy implementation corresponding to `MVCC` from the policy library. Other built-in policies include `Actor` (mailbox serialization), `CRDT` (merge-based), and `Mutex` (blocking lock).

8.4 Policy Library

Table 5 summarizes the four built-in policies and their properties. The key point: *multiple writers are permitted under all policies except Mutex*, which is a strict improvement over Rust's exclusive mutable borrow restriction.

Policy	Multiple Writers	Deadlock-Free	Wait-Free Reads	Distributed
MVCC	✓	✓	✓	✓
Actor	—	✓	✓	✓
CRDT	✓	✓	✓	✓
Mutex	—	—	✓	—

Table 5. Properties of the four built-in proxy policies.

Modifier	Meaning	Default
`+R`	remote / heap / shared / retainable	local (no annot.)
`+M`	mutable	immutable (no annotation)
`+G`	GPU / SIMD / accelerator	CPU (no annotation)
`+C`	cache / near-memory	unconstrained
`+A`	async-surviving (may cross suspension)	suspension-local
`+I`	intentional infinite generator (`w+I`)	finite (.x())

Table 6. Complete settled modifier and domain set. All modifiers are opt-in; the default (no annotation) is the safest and cheapest option in every dimension. There is no $\pm N$ nullability modifier; null safety is operator-based (`??`, `?.`).

8.5 Proxy and the Distributed Potential

Because $+R+M$ objects are *already* mediated by a proxy, the transition from local to distributed execution requires no changes to the program: the proxy implementation is swapped for a remote-dispatch implementation. The type system's guarantee—all mutations go through the proxy—becomes a guarantee that all mutations go through the distributed coordination protocol. This is stated informally as a design property; a full distributed operational semantics is left to future work.

9 The Settled Modifier System: Domains and Refinements

This section consolidates the full settled modifier and modifier system, correcting an earlier formulation that included a $\pm N$ nullability modifier. The settled design handles null safety through operators (`??`, `?.`) rather than type modifiers, and introduces

9.1 Complete Settled Modifier and Domain List

Table 6 gives the complete list of modifiers and domains as settled in the U design.

Unsafe	Safe equiv.	What is bypassed
`arr[i!]`	`arr[i]`	bounds check
`a /! b`	`a / b`	zero-divisor check
`{{expr}}`	`{expr}`	template escaping
`{expr!}`	`{expr}`	template escaping (explicit)
`w+I`	`w` + reachable `b`	infinite-loop guard

Table 7. Settled unsafe convention. Every unsafe operation is visually explicit. A program with no ! markers is safe in all of: bounds, division, template injection, and loop termination.

Remark 9.1 (Why No Nullability Modifier)**.** *An earlier draft of this paper included ±N (nullable/non-null) as a third modifier axis. The settled design rejects this in favor of operator-based null safety:* `??` *(null-coalesce) and* `?.` *(safe-access) handle the two primary null-check patterns without requiring every reference type to carry a nullability annotation.* `None` *is the unique null value—the only multi-letter keyword in U—and the type system distinguishes* `None`*-returning expressions through return type rather than through a reference modifier. This is ergonomically closer to JavaScript/Kotlin's operator approach than to Haskell's* `Maybe` *or Rust's* `Option<T>`*.*

9.2 The Settled Unsafe Convention

Every unsafe operation in U uses a single, consistent visual marker: `!`. Table 7 lists all settled unsafe syntaxes.

Theorem 9.2 (Unsafe Locality)**.** *In any well-typed U program, every violation of bounds safety, division safety, or template injection safety is co-located with a* `!` *marker in the source. A program containing no* `!` *markers satisfies all three properties without runtime traps.*

Proof. By the typing rules: Index-Safe requires either a provable in-bounds condition or a `!` annotation; Div-Safe requires either a provably non-zero denominator or `/!`; Template-Safe requires either context-safe interpolation or `{{}}` / `{!}`. A program with no `!` must satisfy the safe precondition at every site—which is what the safe-by-default forms enforce. □ □

9.3 The Eight-Combination Matrix

Table 8 is exhaustive. The null-check column is independent of the other four: it fires only on $+N$ regardless of ownership or mutability. The sync and proxy columns fire only on $+R+M$ regardless of nullability. The independence of the three axes is what makes the system combinatorially tractable: you reason about each dimension separately and compose.

Remark 9.3 (Shallow Modifiers Are Essential)**.** *All three modifiers are* shallow*: they apply to the reference, not recursively to the object's fields.*

```
node: TreeNode +R -M -N
```

Mods	Semantic contract	Alloc	Null	Sync	Proxy
$-R-M-N$	local immutable non-null	stack	none	none	none
$-R-M+N$	local immutable nullable	stack	on use	none	none
$-R+M-N$	local mutable non-null	stack	none	none	none
$-R+M+N$	local mutable nullable	stack	on use	none	none
$+R-M-N$	shared immutable non-null	heap/ARC	none	none	none
$+R-M+N$	shared immutable nullable	heap/ARC	on use	none	none
$+R+M-N$	shared mutable non-null	heap/ARC	none	proxy	always
$+R+M+N$	shared mutable nullable	heap/ARC	on use	proxy	always

Table 8. The eight core modifier combinations. Each row is a complete semantic and cost specification. The compiler reads this table directly from type annotations; no analysis needed for any column.

This makes `node` *itself shared, immutable, and non-null. It says nothing about the nullability or mutability of* `node.left` *or* `node.right`*; those fields carry their own modifiers. Deep/recursive modifier propagation—as in some ownership systems—leads to combinatorial explosion of annotation burden. Shallow modifiers keep each reference independently reasoned about.*

9.4 The Pit of Success Composes

The default triple $-R-M-N$ is the pit of success in all three dimensions simultaneously. For a programmer who writes no modifier annotations:

- Every function parameter is local, immutable, and non-null. No heap allocation, no ARC, no synchronization, no null check at any call site.
- Every object created locally is mutable and non-null. It may be mutated freely within its scope, will never be null, and disappears when the scope ends.
- Every function is a non-escaping closure. It may be inlined, stack-allocated, or eliminated entirely by the compiler.

The programmer opts *in* to each cost and risk dimension individually: $+R$ to share, $+M$ to make mutable (on a shared object), $+N$ to allow null. Each + annotation is a visible,

conscious decision to accept a cost and responsibility. No cost is ever hidden.

Example 9.4 (Progressive Opt-In).

```
// Zero annotations: all defaults, zero cost, fully safe
f greet(user: User)
    console.log(user.name)

// Opt into nullability: must handle null branch
f greet(user: User +N)
    user +N ? console.log(user.name) : console.log("guest")

// Opt into sharing: config is shared across threads
f render(scene: Scene +R, canvas: Canvas +R +M)
    scene.objects.x(f(obj) = canvas.draw(obj))

// Opt into all three costs: shared, mutable, nullable cache
f withCache(cache: Cache +R +M +N)
    cache +N ? cache.warm() : buildCache()
```

Each additional + annotation is a deliberate choice. The type signature tells a reader exactly what costs and risks the function involves, without reading the implementation.

9.5 Trusting Others' Code: Signatures as Contracts

In most languages, a function signature tells you the types of the parameters but not whether they will be null-checked, retained, or mutated. You must read the implementation—or trust documentation—to know.

In U, the signature is a machine-verified contract:

Example 9.5 (Reading Guarantees from Signatures).

```
f processPayment(
    user:        User        +R -M -N,   // shared, won't mutate, never null
    amount:      Decimal          -N,    // local, non-null (default -R -M)
    transaction: Transaction +R +M -N    // shared mutable, proxy-mediated
): Receipt +R -M -N
```

From this signature alone, a caller knows:

- `user` *will not be modified (the function has only a* $-M$ *view);* `user` *will be retained by ARC for the duration of the call but not stored;* `user` *will not be null-dereferenced.*
- `amount` *is stack-local, non-null, used only within this call.*
- `transaction` *will be mutated—through its proxy, which enforces the declared policy—but the caller can trust the proxy invariant is maintained.*
- *The returned* `Receipt` *is a shared, immutable, non-null value. Safe to pass to any thread, cache, or log.*

None of these guarantees require reading the function body. They are enforced by the type system at compile time and verified at every call site.

This property—*signature completeness*—is a direct consequence of the modifier system. Because all three dimensions of cost and risk (R, M, N) are expressed in the type, the type tells the whole story.

Theorem 9.6 (Signature Completeness). *For any well-typed function* $\mathbf{f}^{\rho}(\overline{x{:}\tau_i^{\rho_i,\mu_i,\alpha_i}})$. *e with return type* $\tau_r^{\rho_r,\mu_r,\alpha_r}$:

1. *No parameter* x_i *with* $\mu_i = -M$ *is mutated within* e.
2. *No parameter* x_i *with* $\rho_i = -R$ *is retained beyond the call.*
3. *No local value with* $\alpha_i = -A$ *is live across a suspension point.*
4. *The returned value satisfies* $(\rho_r, \mu_r, \alpha_r)$*: it is heap-allocated iff* $\rho_r = +R$*; it is proxy-guarded iff* $\rho_r = +R$ *and* $\mu_r = +M$*; it survives suspension iff* $\alpha_r = +A$.

Proof. Each clause follows from the corresponding typing rule: (1) from Field-Write-Local requiring $+M$; (2) from the Stack-Frame Confinement Lemma (Lemma 6.4); (3) from clause 4 of the sharing discipline (Def. 6.3), which prohibits dereference of $+N$ values without a null-check dispatch; (4) from the compilation scheme (Definition 5.1), which allocates to heap iff $\rho = +R$ and wraps in proxy iff $+R+M$. □ □

9.6 Null Safety Without Optional Types

Most languages address null safety with algebraic types: `Option<T>` in Rust, `Optional<T>` in Java, `Maybe T` in Haskell. These require wrapping and unwrapping, adding syntactic overhead even in the common non-null case.

U uses a modifier instead. $-N$ is the default; $+N$ is the exception. The result:

Example 9.7 (Null Handling Without Wrapping Overhead).

```
// Rust equivalent: fn lookup(key: &str) -> Option<User
f lookup(key: S): User +N
    // returns nullable User
// Usage --- no unwrap(), no ?, no match
result: User +N = lookup("alice")
result +N ?
    process(result)      // result is -N in this branch
  : handleMissing()
```

The `??` *dispatch on a* $+N$ *value narrows the type to* $-N$ *in the non-null branch. No wrapper type, no monadic chaining, no explicit unwrap. The compiler elides the null check in any context where the type is already* $-N$.

Example 9.8 (Null Narrowing in Chains).

```
f getCity(user: User +N
    user +N ?
        user.address +N ?
            user.address.city     // type: S -N (narrowe
          : "unknown"
      : "guest"
```

Nested null-check dispatch narrows the type at each level. The compiler emits exactly one null test per `??` *or* `&` *on a* $+N$ *value and none on the narrowed* $-N$ *accesses within the branch. This is the formal content of Theorem 5.5.7.*

Bug class	Eliminated by	How
Dangling reference	Local-default	Stack-frame confinement lemma
Use-after-free	ARC on $+R$	Refcount prevents premature deallocation
Double-free	ARC on $+R$	Single refcount, released once
Dangling suspension-local	$-A$ default	Type error to use $-A$ after `a`
Data race	Proxy on $+R+M$	All mutations serialized by proxy
Hidden mutation	$-M$ default	Type error to write through $-M$ reference
Hidden allocation	Local default	No heap allocation without explicit $+R$
Out-of-bounds access	Safe indexing default	Guard unless `[i!]`
Division by zero	zero-check guard	Guard unless `/!`
Template injection	Escaped default	Raw requires `{{}}`
Infinite loop	Bounded `.x()`	Unbounded requires `w+I`

Table 9. Eleven bug classes structurally eliminated by the settled U design. Each is either a type error or requires an explicit `!` unsafe marker.

9.7 Common Bug Classes Structurally Eliminated

The three-dimensional modifier system eliminates seven major bug classes by making them type errors or unreachable by construction.

Table 9 is the practical consequence of the three theorems. The first three rows follow from ARC-Freedom and the sharing discipline. Row four follows from Theorem 5.5.7 and the sharing discipline. Rows five through seven follow from Race-Freedom, the mutability type rules, and ARC-Freedom respectively. Each entry is "type error" or "structurally unreachable," not "warned about" or "detected at runtime."

10 Settled Language Semantics

This section records the definitively settled design decisions for reference during formalization.

10.1 The Complete Operator Algebra

The separation of logical and bitwise operators is deliberate. Most code uses logical operators; bitwise operations are relatively rare and are made visually distinctive to prevent confusion between `a & b` (guard: evaluate `b` only if `a` is truthy) and `a ~& b` (integer AND: always evaluate both).

Op.	Meaning	Notes
`a | b`	logical OR / fallback	`a ? a : b`; single eval of `a`
`a & b`	logical AND / guard	`a ? b : a`; short-circuits
`a ?? b`	None-coalesce	`a === None ? b : a`
`a?.b`	safe field access	propagates `None`
`!a`	logical NOT	boolean inversion
`~a`	bitwise NOT	integer complement
`a ~& b`	bitwise AND	integer-only
`a ~| b`	bitwise OR	integer-only
`a ~^ b`	bitwise XOR	integer-only
`a ~! b`	bitwise NAND	integer-only

Table 10. U operator algebra. Logical operators (`|`, `&`, `??`) serve as the complete conditional system, replacing `if/else/switch/?:`. Bitwise operators are prefixed with `~` to distinguish them visually from their logical counterparts.

10.2 Evaluation Semantics

All expressions in U follow four evaluation rules:

Definition 10.1 (U Evaluation Semantics).

1. ***Left-to-right:*** *sub-expressions are evaluated strictly left-to-right.*
2. ***Single evaluation:*** *each operand is evaluated at most once, even if it appears in multiple positions (e.g.* `a | b` *evaluates* `a` *once, not twice).*
3. ***Short-circuit:*** `&` *and* `|` *and* `??` *skip the right operand when the left operand determines the result.*
4. ***Explicit transfer only:*** *no value changes domain (local → heap, CPU → GPU, etc.) without an explicit* `.c(domain)` *call. There are no hidden temporaries except compiler-generated optimization temporaries that produce identical observable behavior.*

These four rules together make U's evaluation model entirely predictable: the programmer can read any expression left-to-right and know exactly when each sub-expression is evaluated and exactly when values move between domains.

10.3 Settled Design Decisions Summary

11 Modifier Combinations in Practice

The $\pm R \times \pm M$ modifier space yields four reference quadrants, but the full combinatorial space of interest includes function modifiers, closure capture, parameter passing, field types, collection element types, and return types. This section works through the combinations systematically, showing what each means operationally, what compiler license it grants, and—importantly—which combinations are *type errors* and why that is a feature.

11.1 The Basic Eight Combinations

Area	Decision
Async model	Stackful fibers; `a` is marker and suspend point
Red-blue	Solved: all functions fiber-capable, no coloring
Suspension	Only `+A` values survive suspension
Loops	`.x()` only; `w` for generators
Infinite	`w+I` explicit; plain `w` gets compiler warning
Conditionals	`&`, `|`, `??` only; no `?:` or `if/else`
Iteration	`.x()` always bounded, safe, non-invalidating
Ownership	`.c()` only; no implicit promotion
Unsafe	`!` suffix per-expression (`arr[i!]`, `a /! b`)
Templates	`ctx'...'` auto-escape; `{{expr}}` for raw
Return	`r => value` or incremental `r.field = ...`
Lambdas	`=>` implies lambda; `f` only for named functions
Identifiers	Multi-letter required; single-letter forbidden
Keywords	Single-letter only; `None` the sole exception

Table 11. Settled U design decisions. Each entry is final; none of these are open questions.

Example 11.1 ($-R\,-M\,-N$: Local Immutable Non-Null — Parameter Default).
```
f distance(pt: Point, origin: Point)
    distance = sqrt(pt.x*pt.x + pt.y*pt.y)
```
Both parameters receive $-R\,-M$ by default. The compiler: stack-allocates both, emits no ARC operations, passes by reference (same-frame passing, Theorem 5.5.3), and may vectorize the arithmetic. No annotation required. This is the zero-cost path.

Example 11.2 ($-R\,+M\,-N$: Local Mutable Non-Null — Class Default).
```
acc = Accumulator()          // +M -R by class default
acc.total = acc.total + value
acc.count = acc.count + 1
```
Stack-allocated, no ARC, directly mutated in-place. The compiler may scalar-replace the two fields entirely, keeping them in registers. Cost: identical to a C struct with two integer fields.

Example 11.3 ($+R\,-M\,-N$: Shared Immutable Non-Null — Config).
```
config: AppConfig +R -M = loadConfig()
// passed freely to any number of threads
workers.x(f +R(ctx) = runWith(config, ctx))
```
Heap-allocated, refcounted, but no synchronization on any read (Theorem 5.5.4). Multiple threads may hold references and read concurrently without fences. The compiler: emits no lock instructions for any field access through `config`.

Example 11.4 ($+R\,+M\,-N$: Shared Mutable Non-Null — Proxy-Mediated).
```
cache: Cache +R +M = Cache() [policy: MVCC]
cache.put(key, value)   // dispatches through MVCC proxy
result = cache.get(key) // snapshot read, no blocking
```
Heap-allocated, refcounted, all mutations proxy-mediated. The compiler's exact points-to set for `cache` *is* {`proxy(cache)`}*; no alias analysis needed to confirm the mutation target. Multiple threads may call* `put` *concurrently; the MVCC proxy serializes writes while allowing snapshot reads.*

11.2 Function and Closure Modifiers

The function modifier ρ in $\mathbf{f}^{\rho}$ controls whether the closure itself is stack-resident ($-R$) or heap-resident ($+R$, may escape).

Example 11.5 ($\mathbf{f}^{-R}$: Local Closure — Zero-Cost Lambda).
```
transform = value => value * 2.0
result = data.x(transform)
```
The closure `transform` *is stack-allocated (Theorem 5.5.6). If* `data` *has $-R\,-M$ element type, the entire pipeline compiles to a tight loop with no heap traffic. The compiler may inline the closure and vectorize.*

Example 11.6 ($\mathbf{f}^{+R}$: Escaping Closure — Event Handler).
```
threshold: N = computeThreshold()     // -R by default
handler = f +R(event: Event +R -M)
    event.value > threshold ? alert(event) : ignore(eve
button.onClick(handler)               // handler outlive
```
The $+R$ annotation marks the closure as escaping. The compiler: heap-allocates the closure record, refcounts it, and promotes *the captured* `threshold` *from stack to heap (a copy into the closure's heap record). The promotion is inserted automatically;* `threshold` *remains $-R$ at its declaration site. The promotion cost is paid once at closure creation, not at each call.*

Example 11.7 ($\mathbf{f}^{+R}$ Capturing $+R\,-M$: Shared-Read Callback).
```
config: AppConfig +R -M = loadConfig()
validator = f +R(input: S +R -M)
    input.length < config.maxLength ? valid() : reject(
```
The closure captures `config` *by $+R\,-M$ reference—no copy needed, just an ARC increment. Multiple threads may call* `validator` *concurrently; each reads* `config` *without synchronization. Cost: one ARC increment at closure creation; lock-free reads at every call.*

Example 11.8 ($\mathbf{f}^{+R}$ Capturing $+R\,+M$: Actor Message Handler).
```
inbox: Mailbox +R +M = Mailbox() [policy: Actor]
processor = f +R(msg: Message +R -M)
    inbox.enqueue(msg)   // proxy-mediated, serialized
server.onMessage(processor)
```
The classic actor pattern: an escaping closure holds a $+R\,+M$ reference to its mailbox. All writes go through the Actor proxy, serializing message delivery. Multiple threads may call `processor` *simultaneously; the proxy ensures FIFO ordering.*

11.3 Parameter Passing Combinations

Parameter passing interacts with both the function modifier and the argument's modifier, producing several distinct patterns.

Example 11.9 ($-R\,-M$ parameter, $-R\,+M$ argument: Immutable View).
```
f summarize(data: Stats -R -M)
    r(data.mean + data.stddev)

localStats: Stats -R +M = Stats()
```

```
localStats.mean = computeMean(values)
result = summarize(localStats)    // OK: +M <: -M by subtyping
```

Mutability subtyping ($\tau^{-R,+M} <: \tau^{-R,-M}$) allows a mutable local to be passed where an immutable view is expected. The function receives a read-only reference; within `summarize`*, mutation is a type error. No copy, no ARC. This is the "borrow for reading" pattern, achieved without borrow-checker syntax.*

Example 11.10 (−*R* +*M* parameter: In-Place Algorithm)**.**

```
f normalize(data: [N] -R +M)
    total = data.x(f(x) = x).sum()
    data.x(f(item: N -R +M) = (item / total))
```

The parameter is local and mutable: the function may modify the array in-place. No copy is made at the call site (same-frame passing). The caller retains ownership; the function's mutation is visible to the caller. Useful for in-place sort, normalize, transform.

Example 11.11 (+*R* −*M* parameter: Shared Read, Lock-Free)**.**

```
f render(scene: Scene +R -M, canvas: Canvas +R +M)
    scene.objects.x(f(obj: Object +R -M)
        canvas.draw(obj)    // proxy-mediated write to canvas
    )
```

`scene` *is shared immutable: all reads are lock-free, multiple threads may render different portions of the scene simultaneously.* `canvas` *is shared mutable: all writes go through a proxy (here, the Mutex policy to serialize drawing operations). The type signature makes the concurrency model explicit.*

11.4 Return Type Combinations

The return type modifier determines the allocation strategy for the function's result.

Example 11.12 (−*R* +*M* return: NRVO Construction)**.**

```
f buildReport(data: [Row] -R -M): Report -R +M
    buildReport.title   = "Q3 Summary"
    buildReport.entries = data.x(f(row) = row.format())
    buildReport.total   = data.x(f(row) = row.value).sum()
```

*The return type is −*R* +*M*, so the compiler allocates the* `Report` *in the caller's frame (NRVO by construction). The three assignment statements write directly into the caller's stack slot; no temporary is created, no copy occurs on return. The "return-by-construction" idiom enables this optimization trivially.*

Example 11.13 (+*R* − *M* return: Shared Immutable Factory)**.**

```
f parseConfig(path: S -R -M): AppConfig +R -M
    parseConfig.db   = readString(path, "db")
    parseConfig.port = readInt(path, "port")
```

Return type +R − M: the result is heap-allocated (it will be shared), immutable (no subsequent mutation), and refcounted. Once returned, any number of threads may hold references and read concurrently without synchronization. The factory pattern in its type-system-native form.

Example 11.14 (+*R* +*M* return: Shared Mutable Resource)**.**

```
f openConnection(host: S +R -M): Connection +R +M [poli
    parseConfig.socket = connect(host)
    parseConfig.buffer = Buffer()
```

Return type +R +M: heap-allocated, refcounted, proxy-wrapped. The caller receives a proxy-mediated reference; all operations on the connection are serialized by the Mutex policy. The connection may be passed to multiple threads; each call goes through the proxy.

11.5 Illegal Combinations and the Errors They Prevent

The most revealing tests of a type system are the programs it *rejects*. Each of the following is a type error in λ_U, and each prevents a class of bugs.

Example 11.15 (Storing −*R* in +*R* context: Dangling Reference Prevention)**.**

```
f badFactory(): Widget +R +M
    local: Widget -R +M = Widget()
    r(local)              // TYPE ERROR: -R cannot retu
```

The function attempts to return a stack-allocated value as a heap reference. The type checker rejects this: a −R value cannot inhabit a +R return type without explicit promotion (`.c()` *or a +R ascription triggering heap allocation). In C, this compiles and produces a dangling pointer. In U, it is a type error.*

Example 11.16 (Mutating −*M* reference: Immutability Enforcement)**.**

```
f corrupt(config: AppConfig +R -M)
    config.debug = true    // TYPE ERROR: -M prohibits f
```

The Field-Write-Local *rule requires +M; Field-Write-Shared requires +M and proxy dispatch. Neither applies to a −M reference. This prevents accidental mutation of shared configuration, constants, and interned values.*

Example 11.17 (Direct mutation of +*R* +*M*: Race Prevention)**.**

```
cache: Cache +R +M = Cache()
cache.entries = newMap()    // TYPE ERROR: direct write
                            // requires proxy dispatch
```

The elaborator rewrites field writes on +R +M objects to `[dispatch]` *form. If the proxy flag is not set (i.e., the object was not wrapped via* Proxy-Wrap*), the program is ill-typed. A programmer cannot accidentally bypass the proxy; the type system makes it structurally impossible.*

Example 11.18 (+*R* closure silently promoting −*R* value: No Hidden Allocation)**.**

```
bigBuffer: [N] -R +M = allocLarge(1_000_
handler = f +R(event: Event +R -M)
    process(bigBuffer, event)    // ERROR or WARNING: bi
```

A +R closure that captures a −R value triggers an explicit promotion (heap allocation + copy of `bigBuffer`*). This is visible in the type—the programmer opted into* `f +R`*—and the compiler may emit a warning for large captured objects. The cost is never hidden. In contrast, C++ lambda capture semantics*

silently copy or reference, depending on capture mode, leading to subtle lifetime and performance bugs.

Example 11.19 (−*R* function capturing +*R*+*M*: Prohibition on Implicit Sharing).
```
sharedState: Counter +R +M = Counter()
transform = value => doSomething(value)   // -A closure (local)
    sharedState.increment()       // TYPE ERROR: -R closure
    r(value * 2)                  //   cannot capture +R+M
```

A −*R (non-escaping) closure may not capture a* +*R*+*M value directly, because doing so would make the closure's behavior dependent on shared mutable state in a way that is invisible at the call site. The programmer must explicitly mark the closure* +*R to acknowledge the capture of a shared mutable reference. This prevents the common bug of a "pure-looking" local function that silently reads or writes global state.*

11.6 Nested and Compound Modifier Patterns

Real programs use modifiers on fields, not just on top-level bindings. The interaction of field-level and object-level modifiers produces rich patterns.

Example 11.20 (Mixed-Field Class: Immutable Header, Mutable Body).
```
class Request
    id:      S   +R -M    // shared immutable: safe to log anywhere
    method:  S   +R -M    // shared immutable
    body:    [B] +R +M    // shared mutable: streaming body
    headers: Map +R -M    // shared immutable: set at creation
```

The `id`, `method`, *and* `headers` *fields are* +*R* − *M: they can be read by any number of threads without synchronization. The* `body` *field is* +*R*+*M: streaming data is written concurrently through a proxy. The compiler emits no synchronization for reads of the three* −*M fields; synchronization appears only on writes to* `body`.

Example 11.21 (Array Element Modifiers: The Collection Taxonomy).
```
// Stack array of stack values: SIMD-able, cache-perfect
data: [N -R -M] -R -M

// Stack array of shared immutable refs: reference-counted, no proxy needed
catalog: [Product +R -M] -R +M

// Shared mutable array of local values: concurrent append log
log: [Entry -R +M] +R +M [policy: Actor]

// Shared array of shared mutable items: distributed shards
shards: [Shard +R +M] +R -M
```

The four combinations yield four distinct collection semantics. The compiler generates different code for each: the first compiles to a flat stack-allocated array with SIMD load/store; the second to a pointer array on the stack; the third to a proxy-guarded heap vector; the fourth to a heap-allocated array of proxy references. All four have the same surface syntax; the modifier annotations fully determine the generated code.

12 Modifier Transitions, Subtyping, and Design Patterns

With three independent binary modifier axes, we have not only eight reference types but a rich structure of *transitions* (how one modifier combination becomes another), *subtyping* (which combinations are compatible without conversion), and *design patterns* (which combinations recur in real APIs and what each one means architecturally).

12.1 The Subtyping Cube

The three modifier axes define a partial order on reference types. Two of the three axes admit subtyping; one does not.

Definition 12.1 (Three-Dimensional Subtyping). *The subtyping relation* <: *on modifier triples is:*

$$\tau^{\rho,+M,\nu} <: \tau^{\rho,-M,\nu} \quad [\textsc{Mut-Sub}] \qquad \tau^{\rho,\mu,-N} <: \tau^{\rho,\mu,+N} \quad [\textsc{Null-Sub}]$$

Ownership subtyping (−*R* <: +*R*) *is deliberately excluded.*

The two valid directions are:

- *Mut-Sub: a mutable reference can be used where an immutable one is expected. Immutability is a restriction; the caller simply does not exercise the mutation capability.*
- *Null-Sub: a non-null value can be used where a nullable one is expected. It simply happens to never be null.*

The two valid subtype arrows on μ *and* ν *compose:* $\tau^{\rho,+M,-N} <: \tau^{\rho,-M,+N}$ *—a mutable non-null reference may be used where an immutable nullable one is expected.*

The six valid subtype arrows (one per directed edge of the μ-ν face, for each of the two ρ values) form the following diagram:

$$\begin{array}{ccccc} -R{+}M\ -N & \longrightarrow & -R\ -M\ -N & \longleftarrow & +R{+}M\ -N \\ \downarrow & & \downarrow & & \downarrow \\ -R{+}M{+}N & \longrightarrow & -R\ -M{+}N & \longleftarrow & +R{+}M{+}N \end{array} \qquad \begin{array}{ccc} +R\ -M\ -N \\ \downarrow \\ +R\ -M{+}N \end{array}$$

The bottom-right corner (−*R* − *M* +*N* and +*R* − *M* +*N*) is the most restrictive (fewest capabilities); the top-left (−*R*+*M* −*N* and +*R*+*M* − *N*) is the most permissive. Default parameters (−*R* − *M* − *N*) occupy the bottom-right of the left face—maximally restrictive in capabilities while remaining non-nullable.

12.2 The Conversion Table

Subtyping handles implicit coercions. Explicit operations handle transitions that require runtime work. Table 12 summarizes all modifier transitions and their cost.

Remark 12.2 (Asymmetry of Mutability). *Mutability flows* down *(mutable to immutable is free; immutable to mutable is prohibited). This asymmetry is the formal encoding of the principle "immutability is a restriction you can impose on yourself but cannot impose on data you do not own." It prevents the "const-poisoning" problem of C++, where a* `const` *reference can be cast away; in U, the prohibition is enforced by the type system with no escape hatch.*

From	To	Operation	Cost
$-R\,\mu\,v$	$+R\,\mu\,v$	$+R$ ascription (Let-Shared)	heap alloc + ARC init
$+R\,\mu\,v$	$-R\,+M\,v$	`.c()` shallow copy	stack alloc + field copy
$\rho\,+M\,v$	$\rho\,-M\,v$	implicit (Mut-Sub)	zero
$\rho\,\mu\,-N$	$\rho\,\mu\,+N$	implicit (Null-Sub)	zero
$\rho\,\mu\,+N$	$\rho\,\mu\,-N$	?? / & null-check	one branch + null test
$\rho\,-M\,v$	$\rho\,+M\,v$	*not allowed*	n/a
$+R\,\mu\,v$	$-R\,\mu\,v$	only via `.c()`	(see row 2)

Table 12. All modifier transitions and their cost. Implicit transitions are zero-cost; explicit operations have stated costs. Upgrading immutability ($-M \to +M$) is prohibited: an immutable reference cannot become mutable.

12.3 The "Builder-Freeze" Pattern

One of the most common patterns in safe systems code is building an object locally and then making it immutable and shareable. The modifier transitions make this explicit:

Example 12.3 (Builder-Freeze: Local Mutable to Shared Immutable).
```
// Phase 1: build locally (stack, mutable, non-null)
config: AppConfig -R +M -N = AppConfig()
config.db   = readEnv("DB_URL")
config.port = readEnv("PORT").toInt()
config.debug = false

// Phase 2: freeze (explicit +R ascription -> heap promotion)
frozen: AppConfig +R -M -N = config   // +R annotation triggers Let-Shared
                                      // config is now heap-allocated, ARC'd

// Phase 3: share freely (immutable, no sync needed)
workers.x(f +R(ctx) = startWorker(frozen, ctx))
```

The compiler:

1. *Stack-allocates* `config` *(phase 1, $-R+M-N$).*
2. *On the $+R$ ascription, emits* `halloc` *+ field copy + ARC init.*
3. *For all reads of* `frozen` *($+R-M-N$), emits no fence or lock.*

The cost of immutability is paid exactly once, at the freeze point, not at every read. After freezing, sharing is free.

12.4 Cross-Thread Communication Patterns

The four $+R$ combinations correspond exactly to four standard concurrent communication patterns.

Example 12.4 ($+R-M-N$: Immutable Message Broadcast).
```
event: DomainEvent +R -M -N = buildEvent(payload)
// Broadcast to N subscribers, zero synchronization
subscribers.x(f +R(sub) = sub.notify(event))
```

The event is frozen at creation. All N subscribers may read it concurrently with no fences. Cost: one ARC increment per subscriber at broadcast time, zero synchronization per read. This is the immutable-message pattern used in event-driven architectures.

Example 12.5 ($+R+M-N$ with Actor Policy: Serialized Mailbox).
```
inbox: Mailbox +R +M -N = Mailbox() [policy: Actor
// Many senders, one logical receiver
senders.x(f +R(sender)
    inbox.send(sender.nextMessage())  // proxy serializ
)
inbox.drain(f(msg) = process(msg))    // sequential proc
```

The Actor proxy guarantees that `send` *calls are serialized and* `drain` *sees them in order. The compiler emits proxy dispatch for every write; reads within the actor's own thread are direct. Multiple writers; one logical reader; no deadlock.*

Example 12.6 ($+R+M\,-N$ with MVCC Policy: Concurrent Reads, Serialized Writes).
```
state: AppState +R +M -N = AppState
// Readers: snapshot isolation, no blocking
readers.x(f +R(req)
    snap: AppState -R -M -N = state.c()  // snapshot vi
    renderResponse(req, snap)
)
// Writer: serialized through MVCC proxy
updater.run(f +R()
    state.applyDelta(computeDelta())  // proxy-mediated
)
```

Readers take a shallow copy (snapshot) without locking. The writer goes through the MVCC proxy, which serializes writes and provides snapshot isolation. Readers never block writers; writers never block readers. This is standard MVCC semantics, expressed entirely in U's modifier system.

Example 12.7 ($+R+M-N$ with CRDT Policy: Commutative Writes).
```
counter: GCounter +R +M -N = GCounter() [policy:
// All nodes increment concurrently, no coordination ne
nodes.x(f +R(node)
    counter.increment(node.id)  // commutative, no orde
)
total = counter.value()         // merge is automatic
```

The CRDT proxy accepts concurrent writes from any number of nodes. Because increment is commutative and associative, the proxy needs no serialization—writes from any order produce the same merged result. The compiler still emits proxy dispatch (for the policy invariant), but the policy itself is lock-free. This is the only $+R+M$ pattern where concurrent proxy calls do not require sequential ordering.

12.5 API Design Patterns by Modifier Signature

Real-world APIs map naturally onto specific modifier signatures. A well-designed API's modifier annotations tell the

caller everything they need to know without reading the implementation.

Example 12.8 (File I/O API).
```
f open(path: S +R -M -N): Handle +R +M +N
    // shared mutable handle (writable file), nullable (may fail to open)

f read(handle: Handle +R +M -N, buf: [B] -R +M -N): I -R -M -N
    // reads into local mutable buffer; returns byte count (non-null I)

f close(handle: Handle +R +M -N): B -R -M -N
    // modifies handle state through proxy; returns success bool
```
From signatures alone: `open` *may return null (file not found);* `read` *fills a local buffer and never allocates;* `close` *goes through the handle's proxy (serialized with concurrent reads/writes). The caller needs no documentation beyond the types.*

Example 12.9 (HTTP Client API).
```
f get(url: S +R -M -N): Response +R -M +N
    // shared immutable response body, nullable (network may fail)

f post(url: S +R -M -N, body: [B] +R -M -N): Response +R -M +N
    // body is shared immutable (may be sent multiple times on retry)

f withTimeout(req: Request +R +M -N, ms: I): Request +R +M -N
    // modifies request through proxy; returns same shared mutable request
```
`get` *returns a shared immutable response—safe to cache, pass to multiple handlers, or log, with zero synchronization. The* $+N$ *on returns signals fallibility without an* `Option` *wrapper.* `withTimeout` *takes and returns a* $+R+M$ *request, signaling that the function mutates the request object through its proxy.*

Example 12.10 (Database Query API).
```
f query(conn: DB +R +M -N, sql: S +R -M -N): [Row +R -M -N] +R -M +N
    // shared mutable connection (cursor state), shared immutable rows,
    // nullable result set (query may return no rows or fail)

f transaction(conn: DB +R +M -N, body: F +R(DB +R +M -N)): B -R -M -N
    // body receives a proxy-wrapped connection with transaction policy
    // returns bool (committed / rolled back)
```
The `conn` *parameter is* $+R+M$*: all reads and writes go through the database proxy, which enforces isolation. The result rows are* $+R-M$*: they are shared (may be held by multiple query handlers) and immutable (rows are snapshots, not live views). The nullable result signals that queries can fail or return empty.*

12.6 Modifier Anti-Patterns and Their Diagnostics

Each illegal combination produces a specific, informative type error. Table 13 maps common mistakes to their type error and the invariant being protected.

Anti-pattern	Type error	Protected invariant
Return local as shared	$-R$ return $\neq +R$ type	Stack-frame confinement
Mutate shared directly	direct write to $+R+M$	Proxy mediation
Dereference without null check	field access on $+N$	Null safety
Immutability upgrade	$-M \rightarrow +M$	Immutability permanence
Implicit sharing	$-R$ stored in $+R$ context	ARC-Freedom
Escaping local closure	$-R$ closure returned	Lifetime confinement
Mutating borrowed view	write through $-M$ param	Caller mutation safety

Table 13. Common modifier anti-patterns, the type error they produce, and the invariant the type system is protecting. All are caught at compile time, not at runtime.

12.7 The Red-Blue Problem

Nystrom [19] identified a fundamental asymmetry in languages with explicit `async`/`await`: async functions have a different "color" from synchronous functions. A sync function cannot call an async function without itself becoming async; the annotation propagates upward through the entire call graph, splitting every codebase into two incompatible worlds. Nystrom calls these "red" (async) and "blue" (sync) functions.

The red-blue problem has two concrete costs:

1. **Coloring cost**: every function in the call chain of an async operation must be annotated `async`, even if it does no I/O itself.
2. **Allocation cost**: promise-based async (JavaScript, Python, C# `Task`) heap-allocates a continuation object at every `await` point, regardless of whether the continuation needs to survive the suspension.

U eliminates both costs through a combination of stackful fibers and the `+A` (async-safe) domain.

12.8 Stackful Fibers: Eliminating the Coloring Cost

Definition 12.11 (Fiber Model). *Every U function executes on a* fiber*: a lightweight thread with its own call stack. The fiber scheduler is transparent to function call semantics: calling*

a function looks identical whether or not it will eventually suspend. A suspension point (the a *operator) yields the current fiber to the scheduler and resumes another; the calling fiber's stack is preserved intact on its own stack segment.*

Theorem 12.12 (Red-Blue Freedom). *In U, there is no function-coloring problem: every well-typed function may call every other well-typed function without annotation change. Specifically:*

1. *A function without* a *in its body may call a function with* a*, and the call compiles to a plain fiber-yield without requiring the caller to become* a*-annotated.*
2. *A function with* a *in its body compiles to a resumable fiber; its callers are unaffected.*
3. *No heap allocation is required at a suspension point; the calling fiber's stack is preserved by the fiber scheduler.*

Proof sketch. (1) Because all functions execute on fibers, a yield at an a point yields the *callee's* fiber, not the caller's. The caller's fiber remains on its own stack, blocked waiting for the callee to resume and return. The call semantics from the caller's perspective are identical to a synchronous blocking call. No annotation change in the caller is required.

(2) The a-annotated function's fiber state machine is self-contained; it does not affect the call frame of any caller.

(3) Because fibers have their own stack segments (stackful fibers, not stackless coroutines), no heap continuation object is allocated at a suspension point. The stack segment is preserved by the scheduler; only a scheduler-level context switch is needed. □ □

Example 12.13 (Freedom from Coloring).

```
f processAll(items: [Item])
    items.x(item =>
        result = a fetchDetails(item)   // calls async, but caller unaffected
        store(result)
    )

// caller needs no annotation either
f main()
    items = loadItems()
    processAll(items)    // no 'a' here; fibers handle it transparently
```

In JavaScript, `processAll` *and* `main` *would both require* `async/await`*. In U, neither does.*

12.9 The +A Domain: Eliminating the Allocation Cost

Stackful fibers eliminate coloring but do not by themselves minimize the fiber frame: if the compiler naively saves the entire stack at each suspension point, the memory cost equals the full call depth. The +A domain solves this.

Definition 12.14 (+A Domain). *The* +A *domain (async-safe) marks values that may survive a fiber suspension point. A value without* +A *is* suspension-local*: it must not be live across any* a *suspension in the same fiber.*

Formally, the +A *domain adds a fifth dimension to the modifier space. A reference type is now a tuple* (ρ, μ, ν, α) *where* $\alpha \in \{+A, -A\}$*:*

- $-A$ *(default): value lives on the current fiber's call stack; does not survive suspension.*
- $+A$*: value is preserved across suspension; the compiler saves it into the fiber's resumable frame.*

Example 12.15 (+A Prevents Dangling Suspension-Local References).

```
a loadUser(id: I): User R A
    localBuffer: [B] -A = alloc(1024)      // -A: suspen

    response = a network.fetch(id)         // SUSPENSION

    // TYPE ERROR: localBuffer is -A, cannot be live af
    parseInto(localBuffer, response)       // compiler r

    // CORRECT: use response directly, or promote local
    r => parseResponse(response)
```

The type error prevents a class of bug analogous to returning a pointer to a stack-allocated buffer: the stack frame is gone after the suspension, so a $-A$ *value alive after a suspension point is a dangling reference.*

Example 12.16 (Explicit +A Promotion).

```
a handleRequest(): Respo
    // Needs to survive suspension: explicitly promote
    requestState: Request R A = buildRequest()
// +A: lives in fiber frame

    // Purely local computation: no +A needed
    validation: B -A = validate(requestState)
// stays on stack

    // Suspend: only requestState saved to fiber frame
    responseRaw = a network.send(requestState)
// fiber frame: {requestState}

    r => parseResponse(responseRaw)
```

The fiber frame at the suspension point contains only requestState*. All* $-A$ *values (including* validation*) are discarded; the scheduler saves and restores only the minimal necessary state.*

12.10 Fiber Frame Minimization Theorem

Theorem 12.17 (Fiber Frame Minimization). *Let* f *be a well-typed* a*-annotated function. The compiled fiber state machine for* f *saves and restores at each suspension point exactly the set of* $+A$*-annotated values live at that point—no more, no fewer.*

Formally: let S_i *be the suspension point at position* i *in* f*'s body, and* $\mathsf{Live}^{+A}(S_i)$ *be the set of* $+A$*-annotated values live at* S_i*. The frame saved at* S_i *is* $\mathsf{Live}^{+A}(S_i)$*.*

Proof sketch. The type system enforces (via the suspension-local rule) that no $-A$ value is live across any suspension point. Therefore the only values that must be in the fiber

Modifier	Meaning	Compiler oracle
(none / $-R$)	local, fiber stack	stack alloc, no ARC, no save
`+R`	heap, ARC-managed	heap alloc, ARC increment/decrement
`+M`	mutable	proxy dispatch when combined with `+R`
`+G`	GPU device memory	DMA on `.c(+G)`, no CPU access
`+C`	near-memory	prefetch instructions, cache-line align
`+A`	async-surviving	saved to fiber frame at suspension

Table 14. The six U modifier domains. Each is an optimization oracle: the compiler reads it directly from the type annotation, emitting the corresponding instruction class without analysis. Modifiers compose: `Request +R +A` is heap-managed and survives suspension.

frame are $+A$ values. The compiler need not perform liveness analysis; the $\pm A$ annotations are the liveness oracle. The frame is minimal because every $-A$ value is discarded at suspension by the type rule, and every $+A$ value must be preserved (it is live by construction). □ □

Corollary 12.18 (Async-Zero-Cost for Sync Paths)**.** *A function with no* a *points compiles to a plain call frame with no fiber overhead. A function with* a *points incurs overhead proportional only to* $|\mathsf{Live}^{+A}|$ *at each suspension point, not to total frame depth. In typical I/O-bound code where the fiber frame contains only a request handle and a buffer reference, the overhead is two pointer-sized saves and restores per suspension—orders of magnitude less than a heap-allocated promise object.*

12.11 The Five-Domain Table

The `+A` domain completes the modifier algebra. Table 15 gives all five domains and their combined meaning.

Remark 12.19 (Why Fibers Over Stackless Coroutines)**.** *Stackless coroutines (Rust* `async`*, C++20 coroutines) implement suspension by transforming the function body into a state machine that lives entirely on the heap. This eliminates per-function stack overhead but requires the state machine to capture* all *live variables at each suspension point—the analog of U's fiber frame. The difference is that in stackless systems, the compiler must* infer *which variables are live (via a liveness analysis); in U, the $\pm A$ annotation is the exact liveness oracle, making the transformation provably minimal without analysis.*

Furthermore, stackless coroutines cannot call blocking functions from within an async context (the red-blue problem resurfaces at the blocking boundary), while U's stackful fibers can yield transparently from any call depth.

13 Fibers, the Red-Blue Problem, and the +A Domain

13.1 The Red-Blue Problem

Nystrom [19] identified a fundamental asymmetry in languages with explicit `async`/`await`: async functions have a different "color" from synchronous functions. A sync function cannot call an async function without itself becoming async; the annotation propagates upward through the entire call graph, splitting every codebase into two incompatible worlds. Nystrom calls these "red" (async) and "blue" (sync) functions.

The red-blue problem has two concrete costs:

1. **Coloring cost**: every function in the call chain of an async operation must be annotated `async`, even if it does no I/O itself.
2. **Allocation cost**: promise-based async (JavaScript, Python, C# `Task`) heap-allocates a continuation object at every `await` point, regardless of whether the continuation needs to survive the suspension.

U eliminates both costs through a combination of stackful fibers and the `+A` (async-safe) domain.

13.2 Stackful Fibers: Eliminating the Coloring Cost

Definition 13.1 (Fiber Model)**.** *Every U function executes on a* fiber*: a lightweight thread with its own call stack. The fiber scheduler is transparent to function call semantics: calling a function looks identical whether or not it will eventually suspend. A suspension point (the* a *operator) yields the current fiber to the scheduler and resumes another; the calling fiber's stack is preserved intact on its own stack segment.*

Theorem 13.2 (Red-Blue Freedom)**.** *In U, there is no function-coloring problem: every well-typed function may call every other well-typed function without annotation change. Specifically:*

1. *A function without* a *in its body may call a function with* a*, and the call compiles to a plain fiber-yield without requiring the caller to become* a*-annotated.*
2. *A function with* a *in its body compiles to a resumable fiber; its callers are unaffected.*
3. *No heap allocation is required at a suspension point; the calling fiber's stack is preserved by the fiber scheduler.*

Proof sketch. (1) Because all functions execute on fibers, a yield at an a point yields the *callee's* fiber, not the caller's. The caller's fiber remains on its own stack, blocked waiting for the callee to resume and return. The call semantics from the caller's perspective are identical to a synchronous blocking call. No annotation change in the caller is required.

(2) The a-annotated function's fiber state machine is self-contained; it does not affect the call frame of any caller.

(3) Because fibers have their own stack segments (stackful fibers, not stackless coroutines), no heap continuation object is allocated at a suspension point. The stack segment is

preserved by the scheduler; only a scheduler-level context switch is needed. □ □

Example 13.3 (Freedom from Coloring).

```
// sync function -- NO 'a' annotation
f processAll(items: [Item])
    items.x(item =>
        result = a fetchDetails(item)   // calls async, no color infection
        store(result)
    )

// caller needs no annotation either
f main()
    items = loadItems()
    processAll(items)    // no 'a' here; fibers handle it transparently
```

In JavaScript, `processAll` *and* `main` *would both require* `async`/`await`. *In U, neither does.*

13.3 The +A Domain: Eliminating the Allocation Cost

Stackful fibers eliminate coloring but do not by themselves minimize the fiber frame: if the compiler naively saves the entire stack at each suspension point, the memory cost equals the full call depth. The `+A` domain solves this.

Definition 13.4 (`+A` Domain). *The* `+A` *domain (async-safe) marks values that may survive a fiber suspension point. A value without* `+A` *is* suspension-local*: it must not be live across any* `a` *suspension in the same fiber.*

Formally, the `+A` *domain adds a fifth dimension to the modifier space. A reference type is now a tuple* (ρ, μ, v, α) *where* $\alpha \in \{+A, -A\}$*:*

- $-A$ *(default): value lives on the current fiber's call stack; does not survive suspension.*
- $+A$*: value is preserved across suspension; the compiler saves it into the fiber's resumable frame.*

Example 13.5 (`+A` Prevents Dangling Suspension-Local References).

```
a loadUser(id: I): User R A
    localBuffer: [B] -A = alloc(1024)      // -A: suspension-local

    response = a network.fetch(id)         // SUSPENSION POINT

    // TYPE ERROR: localBuffer is -A, cannot be live after suspension
    parseInto(localBuffer, response)       // compiler rejects this

    // CORRECT: use response directly, or promote data to +A
    r => parseResponse(response)
```

The type error prevents a class of bug analogous to returning a pointer to a stack-allocated buffer: the stack frame is gone after the suspension, so a $-A$ *value alive after a suspension point is a dangling reference.*

Example 13.6 (Explicit `+A` Promotion).

```
a handleRequest(): Response R A
    // Needs to survive suspension: explicitly promote to +A
    requestState: Request R A = buildRequest()
    // +A: lives in fiber frame

    // Purely local computation: no +A needed
    validation: B -A = validate(requestState)
    // stays on stack

    // Suspend: only requestState saved to fiber frame
    responseRaw = a network.send(requestState)
    // fiber frame: {requestState}

    r => parseResponse(responseRaw)
```

The fiber frame at the suspension point contains only `requestState`. *All* $-A$ *values (including* `validation`*) are discarded; the scheduler saves and restores only the minimal necessary state.*

13.4 Fiber Frame Minimization Theorem

Theorem 13.7 (Fiber Frame Minimization). *Let* f *be a well-typed* `a`*-annotated function. The compiled fiber state machine for* f *saves and restores at each suspension point exactly the set of* $+A$*-annotated values live at that point—no more, no fewer.*

Formally: let S_i *be the suspension point at position* i *in* f*'s body, and* $\mathsf{Live}^{+A}(S_i)$ *be the set of* $+A$*-annotated values live at* S_i*. The frame saved at* S_i *is* $\mathsf{Live}^{+A}(S_i)$*.*

Proof sketch. The type system enforces (via the suspension-local rule) that no $-A$ value is live across any suspension point. Therefore the only values that must be in the fiber frame are $+A$ values. The compiler need not perform liveness analysis; the $\pm A$ annotations are the liveness oracle. The frame is minimal because every $-A$ value is discarded at suspension by the type rule, and every $+A$ value must be preserved (it is live by construction). □ □

Corollary 13.8 (Async-Zero-Cost for Sync Paths). *A function with no* `a` *points compiles to a plain call frame with no fiber overhead. A function with* `a` *points incurs overhead proportional only to* $|\mathsf{Live}^{+A}|$ *at each suspension point, not to total frame depth. In typical I/O-bound code where the fiber frame contains only a request handle and a buffer reference, the overhead is two pointer-sized saves and restores per suspension—orders of magnitude less than a heap-allocated promise object.*

13.5 The Five-Domain Table

The `+A` domain completes the modifier algebra. Table 15 gives all five domains and their combined meaning.

Remark 13.9 (Why Fibers Over Stackless Coroutines). *Stackless coroutines (Rust* `async`*, C++20 coroutines) implement suspension by transforming the function body into a state machine that lives entirely on the heap. This eliminates per-function stack overhead but requires the state machine to capture* all *live variables at each suspension point—the analog of U's fiber frame. The difference is that in stackless systems, the compiler must infer which variables are live (via a liveness analysis); in*

Modifier	Meaning	Compiler oracle
(none / $-R$)	local, fiber stack	stack alloc, no ARC, no save
`+R`	heap, ARC-managed	heap alloc, ARC increment/decrement
`+M`	mutable	proxy dispatch when combined with `+R`
`+G`	GPU device memory	DMA on `.c(+G)`, no CPU access
`+C`	near-memory	prefetch instructions, cache-line align
`+A`	async-surviving	saved to fiber frame at suspension

Table 15. The six U modifier domains. Each is an optimization oracle: the compiler reads it directly from the type annotation, emitting the corresponding instruction class without analysis. Modifiers compose: `Request +R +A` is heap-managed and survives suspension.

U, the ±A annotation is the exact liveness oracle, making the transformation provably minimal without analysis.

Furthermore, stackless coroutines cannot call blocking functions from within an async context (the red-blue problem resurfaces at the blocking boundary), while U's stackful fibers can yield transparently from any call depth.

14 Bounded Iteration: `.x()` Safety Guarantees

U has no `for`, `while`, or `do-while` loops. All iteration uses the `.x()` combinator applied to a collection or range. This is not merely an aesthetic choice—it is a source of formal safety guarantees.

14.1 .x() as a Typed Iterator

Definition 14.1 (`.x()` Typing). *For a collection $c : [\tau_e]^{\rho,\mu,\nu}$ and function $f : (\tau_e^{-R,-M,-N}) \xrightarrow{-R} \tau_r$, the expression $c.\mathtt{x}(f)$ has type $[\tau_r]^{-R,+M,-N}$ and is well-typed iff c is finite (i.e., has a statically known or runtime-bounded length). The combinator:*

1. *Iterates f over each element of c in order.*
2. *Never modifies c during iteration (preventing iterator invalidation).*
3. *Produces a result collection on the local stack ($-R+M-N$).*

Theorem 14.2 (.x() Safety). *For any well-typed expression $c.\mathtt{x}(f)$:*

1. ***Termination**: if c is finite, $c.\mathtt{x}(f)$ terminates.*
2. ***Iterator stability**: c is not modified during iteration; no iterator invalidation is possible.*
3. ***Bounds safety**: every element access within f is within bounds by construction—the combinator provides each element directly, not via an index.*
4. ***Local result**: the produced collection is $-R$; no heap allocation occurs for the result unless the caller promotes it.*

Proof sketch. (1) Termination follows from the finite-collection precondition, which the type system enforces (infinite collections require the `w+I` generator syntax, not `.x()`). (2) Iterator stability follows because `.x()` receives the collection by $-R-M$ reference (immutable view); no mutation through this reference is typeable (sharing discipline, item 1). (3) Bounds safety follows because the combinator provides each element directly as a value, never exposing an integer index that could be out-of-range. (4) The result type is $-R+M-N$ by the typing rule; heap allocation requires an explicit $+R$ ascription. □ □

Example 14.3 (What .x() Eliminates).

```
for(i = 0; i <= len; i++) ...          // off-by-one: no
arr[i] = 0; // inside .x() body       // iterator invalid
while(true) arr.x(...)                 // infinite .x():
ptr++; *(ptr)                          // pointer arithme

// U equivalent: all safe by construction
data.x(element =>
    process(element)    // element is -R -M -N, bounded
)
```

14.2 Generators for Intentional Infinite Loops

When an unbounded loop is genuinely intended, the `w` generator provides it explicitly:

```
// Bounded: compiler verifies b reachable
w
    value = nextEvent()
    value.isDone & b          // b = break
    process(value)

// Intentionally infinite: w+I declares intent explicit
w+I
    heartbeat = a ping()
    log(heartbeat)
```

The compiler warns if a `w` loop has no reachable `b`, preventing accidental infinite loops. The `w+I` form declares infinite intent explicitly; no warning is emitted. This is the same `¡` unsafe escape pattern applied to iteration.

15 Conditional Algebra: &, |, and ??

U removes the ternary operator `?:` and replaces it with three short-circuit operators that serve as the complete conditional system. This is not a superficial syntax change—it has formal consequences for evaluation order, optimization, and code uniformity.

15.1 Formal Semantics

Definition 15.1 (Short-Circuit Operators). *Let a and b be expressions. The three conditional operators are:*

$$a \;\&\; b \;\equiv\; a \;?\; b : a \qquad \textit{(if } a\textit{, then } b\textit{, else } a\textit{)}$$

$$a \mid b \;\equiv\; a \;?\; a : b \qquad \textit{(if } a\textit{, then } a\textit{, else } b\textit{)}$$

$$a \;??\; b \;\equiv\; a \texttt{ === None } ? \; b : a \qquad \textit{(if } a \textit{ is None, then } b\textit{, else } a\textit{)}$$

Each operator evaluates a exactly once. b is evaluated only if a does not short-circuit. There is no implicit truthiness conversion; a must be of type B *(Boolean) for* & *and* |*, or of type τ^{+N} for* ??*.*

Example 15.2 (Conditional Algebra in Practice).

```
// Boolean AND-branch: execute b only if a is true
user.isAdmin & renderAdminPanel()

// Boolean OR-fallback: use first truthy value
displayName = user.nickname | user.firstName | "Anonymous"

// Null coalescing: use cached value or compute fresh
result = cache.get(key) ?? compute(key)

// Chained null-safe access (replaces ?. operator)
city = user ?? user.address ?? user.address.city ?? "unknown"
```

Theorem 15.3 (Conditional Completeness). *The three operators $\{\&, |, ??\}$ are expressively complete for all conditional control flow patterns, including:*

1. *Binary true/false branching (& or |).*
2. *Multi-way chained branching (chained &/|).*
3. *Nullable fallback (??).*
4. *Pattern dispatch (chained* `instanceof` *checks via &).*
5. *Short-circuit guard clauses (& for preconditions).*

The removed ?: *operator is derivable as $a \;?\; b : c \;\equiv\; (a \;\&\; b) \mid c$.*

Proof. Each conditional pattern is expressed as shown above. The derivation of ?: follows directly from Definitions: $(a \;\&\; b) \mid c$ evaluates to b when a is true (since $a \;\&\; b = b$, which is truthy when b is truthy) and to c otherwise. This reproduces ?: semantics exactly. □ □

Remark 15.4 (Why Remove ?:). *The ternary operator exists in most languages as a third conditional form alongside* `if/else`*. In U, removing it is consistent with the "one obvious way" principle:* & *and* | *already handle all conditional patterns. The compiler backend generates identical code for &-chains and ?:-chains; there is no performance difference. The gain is cognitive: one precedence model, one evaluation model, one conditional vocabulary.*

16 Hardware Placement Modifiers

Beyond the three modifier axes ($\pm R$, $\pm M$, $\pm N$), U provides a *modifier system* for hardware-level placement. Domains extend the compiler oracle to cover GPU computation and cache-locality optimization.

16.1 Domain Modifiers

Definition 16.1 (Domains). *A domain annotation δ specifies the hardware tier where a value resides:*

Symbol	*Name*	*Meaning*
(none)	*local*	*fiber stack, CPU registers*
+R	*remote*	*heap, ARC-managed, cross-fiber*
+M	*mutable*	*explicitly mutable (surface alias for +M)*
+G	*GPU*	*GPU device memory, SIMD-aligned*
+C	*cache*	*near-memory, cache-resident, prefetch-friendly*

Domain annotations compose with modifier annotations. +R *is the surface syntax for +R;* +M *for +M.* +G *and* +C *are additional dimensions absent from the theoretical $\pm R \times \pm M \times \pm N$ system—they extend the compiler oracle to hardware-level placement decisions.*

Example 16.2 (Domain Annotations in Practice).

```
cpuBuffer: [D]                          // local, stack, CPU
gpuWeights: [D] +G                      // GPU device mem
cacheHot:   [I] +C                      // near-memory, p
sharedLog:  [S] +R +M                   // heap, ARC, mu
```

16.2 Domain-Parameterized Copy

The `.c(domain)` operation transfers ownership between domains:

Definition 16.3 (Domain Copy). *$e.\mathsf{c}(\delta)$ produces a copy of e resident in domain δ:*

$$\frac{\Gamma \vdash e : \tau^{\rho,\mu,\nu} \quad \delta \textit{ is a domain}}{\Gamma \vdash e.\mathsf{c}(\delta) : \tau^{\delta(\rho),\mu',\nu}} \;\textsc{[Domain-Copy]}$$

where $\delta(\rho)$ maps the domain to the corresponding ownership modifier (+R $\mapsto$ +R; +G $\mapsto$ GPU-resident; +C $\mapsto$ cache-resident) and μ' is +M (the copy is always mutable—you copy in order to modify in the new domain).

Example 16.4 (Cross-Domain Copy).

```
cpuMatrix: [D] -R +M -N = loadMatrix(path)
gpuMatrix: [D] +G +M -N = cpuMatrix.c(G)    // explicit
launchKernel(gpuMatrix)

// Local -> Shared: freeze and share
localConfig: AppConfig -R +M -N = buildConfig()
sharedConfig: AppConfig R -M -N = localConfig.c(R)
// heap promotion
workers.x(w => w.configure(sharedConfig))

// Shared -> Local snapshot (MVCC read)
liveState: AppState R +M -N = globalState
snapshot: AppState -R +M -N = liveState.c()
// local snapshot
renderFrom(snapshot)
```

Every cross-domain transfer is a `.c()` *call—explicit, visible, and typed. The compiler knows exactly where each value lives at every point in the program.*

Theorem 16.5 (Modifier Oracle)**.** *For any well-typed expression e with domain annotation δ, the compiler knows at compile time:*

1. *Which memory subsystem e resides in (CPU stack / CPU heap / GPU / cache).*
2. *Whether e requires DMA transfer on copy (CPU ↔ GPU).*
3. *Whether e benefits from prefetch instructions (cache domain).*
4. *Whether e requires ARC operations (remote domain only).*

No profiling, no hardware-performance counter analysis, no speculative optimization is needed. Domain placement is a static fact in the type.

17 Arithmetic Safety and the Unified Unsafe Convention

U's arithmetic operations are safe by default. The compiler either proves safety statically or inserts a runtime guard. When the programmer knows an operation is safe but the compiler cannot prove it, the `!` suffix opts out of the guard.

17.1 Safe-by-Default Operations

Definition 17.1 (Safe Operations)**.** *The following operations are safe by default:*

Op.	*Safe form*	*Unsafe*
Array index	`arr[index]` *(bounds-checked)*	`arr[index!]`
Division	`left / right` *(guarded)*	`left /! right`
Integer overflow	`a + b` *(checked or wrapping)*	`a +! b`
Template interpolation	`{expr}` *(escaped)*	`{{expr}}` / `{expr!}`
Infinite loop	`w` *(requires reachable* `b`*)*	`w+I`

The compiler: (1) proves the check unnecessary via static analysis (eliminates the guard); (2) hoists the check out of loops when provably loop-invariant; (3) inserts a single guard otherwise.

Example 17.2 (Safety in Practice)**.**

```
// Safe: compiler proves index in-bounds from .x() structure
data.x((element, index) =>
    data[index]       // provably in-bounds; guard eliminated
)

// Safe: compiler cannot prove, inserts guard
f lookup(arr: [S], pos: I): S +N
    pos >= 0 & pos < arr.length & arr[pos] | None

// Unsafe: programmer takes responsibility
f fastCopy(src: [B], dst: [B], count: I)
    src[0!]           // no bounds check, programmer gu
```

Theorem 17.3 (Unified Unsafe Convention Soundness)**.** *A program with no* `!` *suffixes is:*

1. *Bounds-safe: no array access is out-of-bounds without a runtime trap.*
2. *Division-safe: no division by zero occurs without a runtime trap.*
3. *Overflow-safe: no unchecked integer overflow occurs.*

Programs with `!` *suffixes may violate these properties at the annotated sites; all other sites remain safe. The* `!` *annotation localizes all unsafe behavior to explicitly marked expressions.*

Remark 17.4 (! as the Universal Unsafe Marker)**.** *The* `!` *suffix plays the same role in U that* `unsafe {}` *blocks play in Rust: it localizes unsafe operations to explicitly marked sites. The difference is granularity: Rust's* `unsafe` *is block-level; U's* `!` *is expression-level. This allows unsafe to be more surgically applied—a single array index in a tight inner loop can opt out of bounds checking while the surrounding code remains safe.*

18 Context-Aware Templates

U includes context-aware tagged templates as a first-class language feature. This extends the compiler oracle to the content domain: the compiler verifies template correctness at the level of the target grammar.

18.1 Template Types

Definition 18.1 (Tagged Template)**.** *A tagged template* `ctx`...`` *has type* `Template[ctx]` *where* `ctx` *is a context identifier (*`html`*,* `sql`*,* `css`*,* `msg`*, etc.). The compiler loads the grammar for* `ctx` *and verifies:*

1. *The template string is syntactically valid in the target grammar.*
2. *All interpolated expressions* `{expr}` *are properly escaped for the target context.*
3. *All interpolated keys in* `msg` *templates exist in the locale bundle.*

Example 18.2 (Template Safety Guarantees)**.**

```
page = html`<h1>{user.name}</h1><p>{content}</p>`
// user.name and content are HTML-escaped; XSS impossib

// SQL: compiler verifies valid SQL, parameterizes {exp
rows = sql`SELECT * FROM users WHERE id = {userId}`
// userId is parameterized; SQL injection impossible

// Raw interpolation requires explicit unsafe marker
raw = html`<div>{{trustedHtml}}</div>`   // {{}} = raw,
```

Theorem 18.3 (Template Safety)**.** *For any well-typed template* `ctx`...`:*

1. *The template string is syntactically valid in the* `ctx` *grammar.*
2. *All* `{expr}` *interpolations are escaped/parameterized for the target context; injection attacks are structurally impossible.*
3. *All* `{key}` *references in* `msg` *templates resolve to existing locale bundle entries; missing-key errors are caught at compile time.*

Raw interpolation `{{expr}}` *opts out of escaping explicitly, following the* `!` *unsafe convention.*

Remark 18.4 (Templates as Domain-Specific Compiler Oracles)**.** *Context-aware templates extend the compiler oracle principle to content: the template tag tells the compiler which grammar to verify against, just as a modifier tells it which optimization to apply. The result is that XSS, SQL injection, and missing-locale errors become compile-time type errors rather than runtime vulnerabilities. This is the "pit of success" principle applied to security-sensitive string operations.*

19 Related Work

WebAssembly and component model. WebAssembly [14] provides a typed low-level bytecode with memory safety guarantees at the module boundary. The WebAssembly Component Model adds interface types for cross-component interoperability. U targets a higher abstraction level: source-language type annotations rather than bytecode types, with the modifier system (`+G`, `+C`, `+A`) providing optimization information that WebAssembly's type system does not represent. U could plausibly compile to WebAssembly as a backend; the modifier annotations would guide the wasm-opt optimization passes in ways the current Component Model type system cannot.

Compiler analysis vs. type-level facts. Traditional compilers use escape analysis [7] to determine heap vs. stack allocation, alias analysis to rule out conflicting writes, and points-to analysis for precise mutation sets. These analyses are approximate, interact with each other, and are among the most expensive in a modern compiler pipeline. U's modifier system replaces all three with exact type-level facts. The closest prior work is effect systems [17], which annotate function types with the effects they may perform. U's $\pm R$ and $\pm M$ modifiers can be seen as a minimal effect system where the effects of interest are *heap allocation*, *sharing*, and *mutation*—precisely the three that drive most compiler analysis complexity. The key distinction is that U's modifiers appear on *values*, not functions, and default to the cheapest option, making the common case annotation-free.

Rust and borrow checking. The most prominent systems language with formal memory safety is Rust [18]. RustBelt [16] gives a formal semantic foundation via a Kripke logical relation in Iris. Rust's borrow checker enforces exclusive mutable access (`&mut`) and shared immutable access (`&`) through non-lexical lifetimes. U differs in two fundamental ways: (1) the $-R$ quadrant requires *no* borrow annotations—exclusivity is the default, not something the programmer must request; and (2) the $+R\,+M$ quadrant permits multiple writers via proxy policies, whereas Rust requires a single `&mut` reference at all times.

Ownership type systems. Clarke et al. [8] survey ownership type systems. The Ownership Types [9] and Universes [12] systems track object ownership for encapsulation rather than memory safety. FractionalPermissions [4] and Capabilities [11] provide read/write permission tracking. U's $\pm M$ modifier is closest to a binary version of fractional permissions; the $\pm R$ modifier is closest to region ownership, but without requiring explicit region annotations.

Region-based memory management. Tofte and Talpin [20] showed that region inference can recover most of the efficiency of manual memory management. MLKit implemented region-based allocation for Standard ML. U's $-R$ objects occupy implicit per-frame regions; the type system ensures no $-R$ value escapes its region, which is formally equivalent to Tofte–Talpin region safety but with a simpler surface language.

Linear and affine types. Linear types [21] enforce use-exactly-once; affine types enforce use-at-most-once. Rust's `&mut` is affine in the reference itself. U's $-R$ modifier is *not* affine—local values may be used multiple times within their scope—which makes it simpler at the cost of permitting local aliasing. The absence of aliasing *across scopes* (enforced by the sharing discipline) is sufficient for the ARC-Freedom and Race-Freedom results.

Null safety in mainstream languages. Kotlin [15] and Swift distinguish nullable (`T?`) from non-nullable (`T`) at the type level, requiring explicit unwrapping. TypeScript's strict null checks and Rust's `Option<T>` take the same approach. U's $\pm N$ modifier achieves the same guarantee without a wrapper type: $-N$ is the default (no annotation), $+N$ is the exception, and the `??`/`&` dispatch on $+N$ values narrows the type to $-N$ in the non-null branch. The difference is ergonomic: the common non-null case requires zero annotation in U, while wrapper-type approaches require explicit `unwrap()`, `!`, or monadic chaining even for non-null values. The formal guarantee is identical: null dereference of a $-N$ value is a type error, not a runtime exception.

***Swift ARC.*.** Swift uses ARC universally for all reference types. This provides safety at the cost of atomic operations on every retain/release. The `isKnownUniquelyReferenced` escape hatch partially addresses this but is not type-checked. U's separation of $-R$ (no ARC) and $+R$ (ARC) at the type

level provides the same optimization as a provable invariant rather than a runtime check.

Capabilities and effects. Gordon et al. [13] and Clebsch et al. [10] use capability systems to control aliasing. The Pony language has six reference capabilities (iso, trn, ref, val, box, tag), each controlling a distinct combination of read/write/alias permissions. U's three-dimensional system ($\pm R \times \pm M$) produces four combinations rather than six. The two cases absent from U are Pony's `trn` (write-unique, read-shared transition capability) and `tag` (opaque, identity-only). `trn` is unnecessary in U because $+R+M$ proxies already handle the transition from exclusive to shared mutable access without a distinct capability mode. `tag` is unnecessary because $+R\ -M$ covers the identity-only pattern. The formal justification for four being sufficient is the Proxy Safety theorem: the proxy abstraction internalizes the concurrency policy that Pony externalizes into the capability lattice. The tradeoff is expressiveness (Pony's lattice is strictly more expressive) vs. cognitive overhead (U's two dimensions are learnable in minutes rather than hours).

Uniqueness types. The Clean language [3] uses uniqueness types to guarantee that certain values have exactly one reference, enabling in-place update without copying. U's $-R$ modifier is weaker than uniqueness—it guarantees non-sharing (no cross-scope reference), not uniqueness within a scope. This is sufficient for ARC-Freedom but does not enable all the in-place update optimizations uniqueness provides. ATS [6] combines linear types with a dependent type system for low-level safety; U targets a similar safety level with substantially simpler user-facing annotations.

20 Discussion

Mechanization. The proofs in this paper are not mechanized. We identify the specific targets for mechanization: (1) The Type Soundness theorem (Progress + Preservation) is a standard structural induction and would translate directly to Coq using the locally nameless representation [5]. (2) The Race-Freedom theorem requires formalizing the concurrent semantics; Iris [16] is the natural target, using the same separation-logic infrastructure as RustBelt. (3) The Fiber Frame Minimization theorem requires the concurrent operational semantics plus the Proxy Non-Interleaving axiom (Axiom 4.6); the axiom is the one assumption that would need to become a proof obligation against a concrete scheduler. (4) The Modifier-Optimization Correspondence theorem would be proved in Coq against the TAL_U compilation scheme, which is already defined precisely enough to formalize. We regard mechanization of items (1) and (4) as near-term work given the proof structure already established.

What U does not prove. The $-R$ quadrant does not prevent local aliasing within a scope: two local variables may refer to the same $-R$ object. This is a known gap relative to Rust's exclusive `&mut`. Local aliasing does not produce races (by Race-Freedom), but it does permit unexpected mutation when two local aliases are updated. Detecting this at compile time would require alias analysis or a restricted form of uniqueness typing; we leave this as a known limitation and a direction for future work.

Shallow vs. deep copy. The `.c()` operation is deliberately shallow. Deep recursive copying would require a `Clone`-like trait system; shallow copying is predictable and cheap. Programs needing deep copies express them explicitly by walking the object graph. This is a deliberate design choice for *predictability*: the programmer always knows exactly what was copied.

Generator formalization. Generators—the primary iteration primitive—are omitted from λ_U. They can be encoded as coroutines with an explicit continuation type $\tau \to (\tau \times (\mathtt{I} \xrightarrow{-R} \tau))$; the $-R$ modifier on the continuation ensures it does not escape the iteration scope. We leave the full formalization to future work.

Async unification. Because generators support suspension, state preservation, and coordination, they subsume extttasync/ extttawait in many cases. Combined with $+R$ proxies for cross-thread communication, U may not require a separate async primitive. The formal relationship between generator semantics and async calculi (e.g., the π-calculus) is left to future work.

Return-by-construction and NRVO. U's assignment-to-function-name return style (`compute` $= v$) enables named return value optimization (NRVO) by construction: the compiler can allocate the return value in the caller's frame and have the callee write directly into it, eliminating copies entirely. For $-R$ objects, this combines with stack allocation to produce zero-copy, zero-allocation construction of complex values. A formal treatment via a destination-passing transformation is future work.

Scope and threats to validity. The results in this paper are theoretical: we prove properties of the formal calculus λ_U, not of an implemented compiler. Three caveats are worth stating explicitly. (1) *Generics*: λ_U has no type parameters. Real programs require parametric polymorphism; we expect the modifier system to compose cleanly with a System F-style type parameter mechanism, but this is not proved here. (2) *Proxy Non-Interleaving* (Axiom 4.6): this axiom is assumed, not derived. Concrete proxy implementations (MVCC, Actor, Mutex) satisfy it by construction; the axiom is a specification requirement on proxy implementors. (3) *Fiber Frame Minimization* assumes the compiler correctly identifies `+A`-annotated liveness; if the programmer under-annotates `+A`, the compiler will reject the program (a type error); if they

over-annotate, the frame is larger than minimal but still correct. These caveats do not affect the soundness of the proved theorems; they bound the scope of the results.

Open design questions. Several language areas are intentionally unresolved in this paper: generics and type-parameterized containers; interface/trait system; error handling and propagation; module and import system; object layout ABI; deterministic destruction and finalizers; and unsafe/FFI escape hatches. The ownership type system presented here is designed to be orthogonal to all of these; we expect each to compose with the $\pm R/\pm M$ modifier system without requiring changes to the metatheory.

21 Expression System, Type Inference, and Error Model

This section covers the remaining settled language features: expression-oriented evaluation, type inference, the complete interpolation syntax, safe access chains, the error model, and the memory lifecycle.

21.1 Expression-Oriented Blocks

Every construct in U is an expression. Blocks delimited by indentation or parentheses evaluate to their final expression:

```
result = (
    validated = validate(input)
    transformed = transform(validated)
    transformed.normalize()    // block value = last expression
)
```

Single-line blocks use commas:

```
result = (validate(input), transform(it), it.normalize())
```

This unifies statement and expression contexts: there is no distinction between "a block that returns a value" and "a block that performs side effects." Every block has a type; the type of a void block is `None`.

21.2 Lambda Syntax and the => Operator

Named functions use `f`; anonymous functions (lambdas) use `=>` directly—no `f` keyword needed:

```
// Named function
f square(value: N): N
    r => value * value

// Lambda: => implies anonymous function
transform = value => value * 2

// Multi-expression lambda uses block
process = item => (
    cleaned = item.trim()
    cleaned.lowercase()
)

// .x() with lambda
results = data.x(item => item.score > threshold &item)
```

The `=>` in a lambda context is *not* the same token as `r =>` (return assignment); the parser disambiguates by context—`r =>` appears only inside a function body where `r` is bound.

21.3 Type Inference

U infers types from initialization expressions. The type rules have a straightforward inference direction: the right-hand side determines the type; the left-hand side is a pattern.

Definition 21.1 (Type Inference Defaults)**.** *When no type annotation is present:*

1. *A numeric literal without decimal point infers* `I` *(integer).*
2. *A numeric literal with decimal point infers* `N` *(number / float).*
3. *A string literal infers* `S`.
4. *An object literal* `{f: e, ...}` *infers a structural record type with the field types inferred from the values.*
5. *A lambda* `x => e` *infers a function type from the body.*
6. *A* `None` *literal infers* τ^{+N} *where* τ *is determined by context (or* `Any +N` *if context is absent).*

The modifier defaults ($-R-M-N$) apply to all inferred types.

Example 21.2 (Type Inference in Practice)**.**

```
count = 0
// I -R +M -N  (mutable local integer)
ratio = 3.14          // N -R -M -N  (immutable local
name  = "Alice"       // S -R -M -N
user  = { name: "Bob", age: 30 }   // {name:S, age:I} -
double = x => x * 2   // F(N):-R  (inferred from usage)
```

21.4 Width Specifiers

Primitive types carry optional width specifiers: `I8`/`I16`/`I32`/`I64` (signed integers), `U8`–`U64` (unsigned), `N32`/`N64` (floating-point), `D32`/`D64`/`D128` (decimal), `B8` (byte). Width specifiers are orthogonal to the modifier system: `I32 +R -M -N` has all the same guarantees as `I +R -M -N`, just with a fixed 32-bit storage width.

21.5 Range Syntax

The range literal `start..end` produces an iterable range value:

```
(1..100).x(index => process(index))     // [1, 100)
exclusive end
(1..=100).x(index => process(index))    // [1, 100]
inclusive end
(0..data.length).x(i => data[i])        // index range
```

Ranges are $-R-M-N$ by default (local, immutable, non-null). The compiler proves bounds safety for `.x()` over a range: since `start..end` is bounded, `.x()` over it terminates (Theorem 14.2).

21.6 Safe Access Chains and Null Propagation

The `?.` operator propagates None through access chains without explicit null checks at each step:

Definition 21.3 (Safe Access Type Rule). *For $e : \tau^{\rho,\mu,+N}$ and field f of type $\tau_f^{\rho',\mu',-N}$:*

$$\frac{\Gamma \vdash e : \tau^{\rho,\mu,+N}}{\Gamma \vdash e?.f : \tau_f^{\rho',\mu',+N}} \; [\textsc{Safe-Access}]$$

If e is None*, the entire chain short-circuits to* None*. The result type carries $+N$ regardless of whether τ_f is $-N$ or $+N$: nullability propagates through the chain.*

Example 21.4 (Null Propagation Chains).

```
// Each ?. propagates +N; final type is S +N
city: S +N = user?.address?.city

// Terminate chain with ?? to get -N
cityName: S -N = user?.address?.city ?? "unknown"

// Compiler emits exactly two null tests, not three
// (user and address; city's own -N-ness means no test there)
```

21.7 Complete Interpolation Syntax

U templates support four interpolation forms, each with a distinct safety level:

Form	Safety	Meaning
`{expr}`	auto-escaped	escaped/parameterized for context
`{{expr}}`	raw (unsafe)	bypasses escaping; reviewer visible
`{expr!}`	raw explicit	same as `{{...}}`, `!` convention
`{expr?}`	optional	inserts nothing if expr is None

Table 16. Template interpolation forms. The default `{expr}` is always safe; the `!` forms follow the unified unsafe convention. The `?` form handles optional content without a null check.

Example 21.5 (Optional Interpolation).

```
// {expr?} inserts nothing if expr is None — no explicit check needed
page = html`
    <h1>{title}</h1>
    <p class="subtitle">{subtitle?}</p>
    <img src="{avatar?}">
`
```

The `?` form is syntactic sugar for `expr ?? ""` *within a template context—safe, auto-escaped, and null-handling without ceremony.*

21.8 Localization as a Compile-Time Guarantee

The `msg` template context extends Template Safety to internationalization:

Theorem 21.6 (Localization Completeness). *For any well-typed* `msg‘...‘` *template:*

1. *Every* `{key}` *reference resolves to a key present in the declared locale bundle at compile time.*
2. *Every placeholder in the locale string has a corresponding* `{expr}` *in the template.*
3. *Missing keys, extra placeholders, and type mismatches between expressions and locale format strings are compile-time errors.*

Example 21.7 (Compile-Time i18n Verification).

```
// Locale bundle: e
msg`{greeting, name: user.name}`   // OK: key exists, p
msg`{farewell}`                    // OK: no placehold
msg`{welcome}`                     // COMPILE ERROR: k
msg`{greeting}`                    // COMPILE ERROR: p
```

No existing language provides this guarantee. Internationalization bugs (missing translations, mismatched placeholders) are runtime errors in every other framework; in U they are type errors.

21.9 Error Propagation via e and y

The `e` keyword throws an error that propagates up the fiber:

```
f divide(numerator: N, denominator: N): N
    denominator == 0 & e DivisionError("denominator is
    r => numerator / denominator
```

Errors propagate through the fiber's call stack until caught. Because fibers are stackful, error propagation is synchronous regardless of whether the fiber has suspended—no `Promise.catch` chains, no `Result<T, E>` unwrapping at every call site.

The `y` keyword yields a value from a `w` generator:

```
f naturals(): I    // generator: yields I values
    current = 0
    w+I
        y current
        current = current + 1
```

Generators and errors compose: a generator can `e` to signal exhaustion or invalid state; callers receive the error through the same propagation mechanism as non-generator code.

21.10 Memory Lifecycle and Per-Domain GC

U's memory model is deterministic for the common case and has a background collector only for the exceptional case:

Definition 21.8 (Memory Lifecycle by Domain).

1. ***Local*** *(−R): freed at scope exit. No runtime overhead; the compiler inserts destructor calls at the closing brace of the binding scope. Deterministic, zero-cost.*
2. ***Remote*** *(+R): freed when the reference count reaches zero (ARC). Deterministic for acyclic object graphs.*
3. ***Cycles****: objects in +R reference cycles are collected by a per-domain background cycle collector. Cycle collection is bounded in scope (per +R domain, not global) and triggered only when the ARC detects a candidate cycle*

(reference count > 0 but no path from a stack root). This is not a global GC; it is a surgical cycle detector with bounded pause time.

4. ***GPU (+G):*** *freed when the last* `+G` *reference drops; the runtime issues a DMA deallocation. GPU memory is never collected by the cycle detector.*

Remark 21.9 (Why "per-domain/background only")**.** *The full GC machinery runs only on* `+R` *objects that are provably in a cycle. The overwhelming majority of +R objects are acyclic (trees, lists, value graphs); these are freed by ARC with zero GC involvement. The cycle detector activates only when an ARC decrement leaves a non-zero count on a heap object with no live stack root—a condition the runtime detects cheaply via a "purple" candidate list (the tri-color concurrent cycle detection algorithm [2]). −R objects never enter the collector at all. The result: most programs see zero GC pauses; programs with deliberate cycles see bounded, local pauses.*

21.11 `d` Classes, `t` Self, and the Class Model

Classes are declared with `d` (define):

```
d Point
    x: N
    y: N

    f distance(other: Point): N
        dx = t.x - other.x
        dy = t.y - other.y
        r => (dx*dx + dy*dy).sqrt()
```

`t` is the self reference inside methods (analogous to `this` in Java/JS, `self` in Python/Swift). There is no separate `struct` keyword; a `d` class without shared instances is stack-allocated (−*R* default) and has the same layout as a C struct. Methods are regular `f` functions with an implicit `t: ClassName -R -M -N` first parameter.

22 Comparison with Existing Languages

U targets the same design space as several widely-used languages. We compare U against five: Rust, Go, JavaScript, Python, and C++. Table 17 gives a high-level summary; this section provides detail on the most important dimensions.

22.1 U vs. Rust

Rust is U's closest peer in the systems safety space. Both languages provide memory safety without a garbage collector, zero-cost abstractions, and data-race freedom. The differences are architectural:

Ownership model. Rust's borrow checker enforces exclusive mutable access via lifetime annotations and non-lexical lifetimes. A function that borrows a value mutably must prove, statically, that no other reference to the same value exists for the duration of the borrow. This provides a strong guarantee but imposes substantial annotation burden and cognitive overhead: lifetime parameters (`'a`), `Pin`/`Unpin`, the `Rc`/`Arc`/`RefCell` proliferation, and the sharp distinction between `&T` and `&mut T` are all manifestations of the same underlying complexity.

U replaces the borrow checker with the modifier algebra. The −*R* default provides exclusivity by construction (no other reference exists because sharing is opt-in via `+R`). The +*M* modifier on shared objects routes all mutations through a proxy, which enforces safety by policy rather than by static exclusion. The tradeoff: U permits multiple concurrent writers (via MVCC/CRDT proxy policies) that Rust prohibits without unsafe code.

Property	U	Rust	Go	JS	Py	C++
Memory model	domains	borrow	GC	GC	GC	manual
Null safety	type err	`Option`	nil	undef	None	nullptr
Data races	type err	type err	runtime	runtime	GIL	UB
Async model	fibers	stackless	goroutines	promises	asyncio	coro
Red-blue	solved	partial	solved	present	present	present
0-cost closures	yes	yes	no	no	no	partial
GPU support	domain	unsafe	no	no	partial	CUDA
Bounds chk	default	default	runtime	n/a	n/a	UB
Null deref	type err	type err	runtime	runtime	runtime	UB
Template safety	compile	no	no	no	no	no
Single eval	yes	no	no	no	no	no

Table 17. Feature comparison across six languages. "type error" means the property is enforced statically; "runtime" means a runtime exception may occur; "UB" means undefined behavior.

Async model. Rust's `async`/`await` uses stackless coroutines transformed into `Future` state machines. This solves the allocation problem (state machines are stack-allocated) but does not solve the coloring problem: `async fn` and `fn` are distinct types, and the async annotation propagates through call chains. Blocking inside an async context requires explicit `spawn_blocking`.

U solves both problems: stackful fibers eliminate coloring (any function may call any other); the `+A` domain minimizes the fiber frame to only suspension-surviving values, recovering the allocation efficiency of stackless systems.

Null safety. Rust uses `Option<T>` to represent nullable values, requiring explicit `unwrap()`, `?`, or pattern matching at every use site. U's −*N* default achieves the same guarantee with no wrapping overhead; `??` and `&` serve as the null-dispatch operators.

22.2 U vs. Go

Go shares U's goal of simplicity and uses goroutines—a fiber-like model—for concurrency. The key differences:

Memory safety. Go uses garbage collection. This simplifies ownership but introduces pauses, increased memory usage, and the inability to reason statically about deallocation time. U's modifier system provides deterministic deallocation (local values are freed at scope exit; shared values when the last +R reference drops) without GC pauses.

Nil safety. Go still has nil pointer dereferences at runtime. Any pointer type in Go is nullable by default; nil checks are idiomatic but not enforced. U's $-N$ default makes null dereference a compile-time type error.

Type system. Go's interface system is structural but its type system does not track ownership or mutability. Two goroutines can hold the same map reference and race on it; the race detector catches this at runtime (with coverage limitations) rather than at compile time. U's +A domain means only +A values enter goroutine-equivalent fiber frames; the type system prevents races structurally.

22.3 U vs. JavaScript

JavaScript is U's closest peer in ergonomic design—both languages support first-class closures, functional iteration, and expressive string templates. U can be seen as "what JS would look like if designed with safety and performance as first-class goals."

The async gap. JavaScript's async/await introduced the red-blue coloring problem [19] at scale: any function that needs to call an async API must itself become async, and this annotation propagates through entire codebases. U's fiber model eliminates this: the a keyword marks suspension points, but caller functions are not annotated. A U codebase has no equivalent of "async all the way up."

Allocation model. Every closure in JavaScript is heap-allocated. U's $-R$ default stack-allocates closures that do not escape; heap allocation is explicit via +R. In tight inner loops—the performance-critical path in data processing—U closures passed to .x() compile to zero-allocation iterations.

Template safety. JavaScript's template literals provide string interpolation but no grammar verification. Inserting user input into an HTML template literal is an XSS vulnerability waiting to happen. U's html‘...‘ template auto-escapes all {expr} interpolations at compile time; injection is structurally impossible without an explicit {{expr}} unsafe marker.

Type system. JavaScript has no static type system (TypeScript adds one externally). U's three-dimensional modifier system is statically verified; the eight-combination table (Table 8) is checked at compile time, not at runtime.

22.4 U vs. Python

Python shares U's readability goals and functional iteration style. Both use significant whitespace; both favor expressive, composable code. The gaps are performance and safety:

Performance. Python's global interpreter lock (GIL) prevents true parallelism in CPU-bound code. GC and dynamic dispatch add constant-factor overhead. U compiles to native code with no GC and no GIL equivalent; the +G domain enables GPU-accelerated computation with the same syntax as CPU computation. A Python NumPy program and a U program with +G array annotations solve the same problem; the U version compiles to CUDA or SIMD without a separate library.

Safety. Python raises AttributeError on None attribute access, IndexError on out-of-bounds access, and ZeroDivisionError on division by zero—all at runtime. U makes all three type errors or compile-time guards; the ! suffix opts out explicitly.

Closures and iteration. Python's list comprehensions and map/filter/reduce serve the same purpose as U's .x(). The difference: Python's comprehensions allocate a new list; U's .x() returns $-R+M$ (stack-allocated) by default. Python lambda has no equivalent of U's zero-cost local closure; all Python closures are heap objects.

22.5 The Design Space U Occupies

Table 17 shows three columns where U is strictly better than every other language surveyed.

Red-blue / coloring: Go and U both solve this; Rust, JavaScript, Python, and C++ do not. But U additionally provides Fiber Frame Minimization (Theorem 13.7), which Go's goroutines do not: Go saves the entire goroutine stack, while U saves only +A-annotated values.

Template and i18n safety: U is the only language in the table where injection attacks *and* missing locale keys are compile-time type errors. Every other language relies on library conventions, linters, or runtime checks—all bypassable. The Localization Completeness Theorem (Theorem 21.6) is a novel guarantee that no other language provides at any level.

Single-evaluation guarantee: U is the only language that formally guarantees each operand in a conditional expression is evaluated exactly once (Definition 10.1). This eliminates a class of subtle bugs (double evaluation of side-effectful expressions) that affects all other languages in the table.

U targets the intersection that no existing language occupies: *Go's concurrency ergonomics* (no function coloring), *Rust's memory and race safety* (proved, not GC-managed), *C's performance* (zero-annotation path is zero-overhead), and *compile-time injection safety* (unique to U). The mechanism is the modifier algebra: annotations that are simultaneously

safety proofs and compiler oracles, with defaults chosen so the common path requires none.

23 Conclusion

The Central Insight

Every type annotation in U makes a claim to two audiences at once. To the programmer: "this value cannot be null, cannot escape this scope, cannot be mutated through this reference." To the compiler: "do not check for null here, allocate on the stack, skip the synchronization." These are not two separate claims—they are one claim, stated once, that serves two masters. We call this *compiler-transparent ownership*, and we have formalized it as a sixteen-theorem theory of a multidimensional modifier algebra.

What Changed

Before this work, language designers faced a perceived tradeoff: make annotations *for programmers* (safety, ergonomics) or *for compilers* (analysis, optimization). Rust chose programmer safety at the cost of annotation complexity. Go chose ergonomics at the cost of frame-size overhead. C chose performance at the cost of safety. U's contribution is showing the tradeoff is false. The *same* annotation that makes a program safe also makes it trivially optimizable—not as a coincidence of design, but as a *theorem*: the Modifier-Optimization Correspondence Theorem (Theorem 5.5) proves that each annotation in the modifier algebra is simultaneously a safety invariant and a sound compiler transformation license. Safety follows as a corollary of optimization transparency.

Three Problems Closed

We identified three problems that have resisted clean solutions across all major languages and proved formal closures for each.

The function-coloring problem [19]— every async/await language forces async annotations to propagate through call chains, splitting every codebase into two incompatible worlds—is closed by the Red-Blue Freedom Theorem (Theorem 13.2) and Fiber Frame Minimization Theorem (Theorem 13.7) together. U achieves Go's ergonomics (no coloring) and Rust's efficiency (minimal suspension frame) simultaneously, with formal proof. Neither language achieves both.

The injection safety problem—SQL injection and XSS arise from untyped string interpolation and affect production systems across the entire industry—is closed by the Template Safety Theorem (Theorem 18.3). Injection is a compile-time type error in U, not a runtime vulnerability that library conventions may or may not prevent. No other language provides this guarantee.

The i18n correctness problem—missing translation keys and placeholder mismatches silently produce runtime errors in internationalized applications—is closed by the Localization Completeness Theorem (Theorem 21.6). Missing keys and unsatisfied placeholders are compile-time errors. No other language or framework provides this guarantee.

What This Enables

A programmer writing idiomatic U with no modifier annotations gets, by construction and by the theorems proved here: memory safety, race freedom, null safety, async correctness, injection safety, i18n correctness, and bounded iteration. Seven properties. No annotations. No garbage collector. No borrow checker. No `Option` unwrapping. No async/await coloring. No runtime injection failures. No missing-translation crashes.

Opting into any cost—shared ownership, mutability, nullability, GPU placement, async survival—requires an explicit `+X` annotation. Every cost is visible; every risk is opted into. This is the dual pit of success: the easy path for programmers is the correct path, and the easy path for compilers is the optimal path.

Looking Forward

Two directions are open. Mechanization—the Coq formalization of Type Soundness against the locally nameless representation [5], and the Iris-based proof of Race-Freedom—will close the gap between the present proof-sketch treatment and machine-verified certainty. Generics—type-parameterized containers and functions—will extend the multidimensional modifier algebra to the full expressiveness real programs require, and we expect the orthogonality of the modifier dimensions to compose cleanly with a System F core.

The modifier algebra presented here is the foundation. A well-typed U program cannot dangle a reference, race on shared data, dereference null, accidentally leak async state across a suspension boundary, produce an injection vulnerability, or crash on a missing translation key. Not because the programmer was careful. Because the type system made carelessness a compile error.

References

[1] Brad Abrams. 2003. The Pit of Success. Blog post. https://learn.microsoft.com/en-us/archive/blogs/brada/the-pit-of-success.

[2] David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*. 207–235.

[3] Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Type Inference. In *Proceedings of the 8th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP)*. 189–206.

[4] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Proceedings of the 10th International Symposium on Static Analysis*. 55–72.

[5] Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (2012), 363–408.

[6] Chiyan Chen and Hongwei Xi. 2005. Combining Programming with Theorem Proving. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 66–77.

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of*

the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). 1–19.

[8] Dave Clarke, Tobias Wrigstad, and Johan Östlund. 2013. Ownership, Uniqueness, and Immutability. *Lecture Notes in Computer Science* 7850 (2013), 1–22.

[9] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 48–64.

[10] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE)*. 1–12.

[11] Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 262–275.

[12] Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. In *Journal of Object Technology*, Vol. 4. 5–32.

[13] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 21–40.

[14] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Catlin Bastian. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 185–200.

[15] JetBrains. 2016. Null Safety in Kotlin. https://kotlinlang.org/docs/null-safety.html.

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the ACM on Programming Languages (POPL)*, Vol. 2. 66:1–66:34.

[17] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 47–57.

[18] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[19] Bob Nystrom. 2015. What Color is Your Function? https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/.

[20] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176.

[21] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. 347–359.

A Full Typing Rules

$$\frac{}{\Gamma \vdash c : \beta}\ [\textsc{Const}] \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ [\textsc{Var}]$$

$$\frac{\Gamma \vdash e_i : \tau_i \text{ for all } i \quad \text{modifiers unspecified} \Rightarrow \rho{=}{-R}, \mu{=}{+M}}{\Gamma \vdash \{\overline{f_i = e_i}\} : \{\overline{f_i{:}\tau_i}\}^{\rho,\mu}}\ [\textsc{Obj}]$$

$$\frac{\Gamma \vdash e : \{\overline{f{:}\tau_f}\}^{\rho,\mu} \quad f_j \in \overline{f}}{\Gamma \vdash e.f_j : \tau_{f_j}}\ [\textsc{Field-Read}]$$

$$\frac{\Gamma \vdash e : \{\overline{f{:}\tau_f}\}^{-R,+M} \quad \Gamma \vdash e' : \tau_{f_j}}{\Gamma \vdash e.f_j := e' : \{\overline{f{:}\tau_f}\}^{-R,+M}}\ [\textsc{Field-Write-Local}]$$

$$\frac{\Gamma \vdash e : \{\overline{f{:}\tau_f}\}^{+R,+M} \quad \Gamma \vdash e' : \tau_{f_j}}{\Gamma \vdash e\ [\text{dispatch}]\ (e') : \{\overline{f{:}\tau_f}\}^{+R,+M}}\ [\textsc{Field-Write-Shared}]$$

Note: direct write ($e.f_j := e'$) on a $+R+M$ object is *not typeable*. The elaborator rewrites surface field assignments to [dispatch] form when the target has $+R+M$ type.

B Proofs

B.1 Substitution Lemma

Lemma B.1 (Substitution). *If* $\Gamma, x{:}\tau' \vdash e : \tau$ *and* $\Gamma \vdash v : \tau'$, *then* $\Gamma \vdash e\{v/x\} : \tau$.

Proof. Standard induction on the derivation of $\Gamma, x{:}\tau' \vdash e : \tau$. All cases are routine; the sharing discipline ensures no modifier conflict arises during substitution (a $-R$ value substituted for a $-R$ variable remains $-R$; a $+R$ value cannot be substituted for a $-R$ variable by the sharing discipline). □

B.2 Canonical Forms Lemma

Lemma B.2 (Canonical Forms). *If* e *is a closed value with* $\emptyset \vdash e : \tau$, *then:*

- *If* $\tau = \beta$, *then* e *is a base constant.*
- *If* $\tau = \{\overline{f{:}\tau_f}\}^{-R,\mu}$, *then* e *is a stack location.*
- *If* $\tau = \{\overline{f{:}\tau_f}\}^{+R,\mu}$, *then* e *is a heap location.*
- *If* $\tau = (\overline{\tau}) \xrightarrow{\rho} \tau_r$, *then* e *is a function closure.*

Proof. By inspection of the value grammar and typing rules. The key observation: Let-Local produces stack locations of $-R$ type, Let-Shared produces heap locations of $+R$ type, and the two cases are mutually exclusive by the sharing discipline. □

B.3 Full Proof of ARC-Freedom

We expand the proof sketch of Theorem 7.1.

Full proof of Theorem 7.1. Let e be closed and well-typed with $\emptyset \vdash e : \tau^{-R,\mu}$. We prove by induction on the length of the evaluation sequence that no step allocates a heap location or performs an atomic operation.

Base case ($k = 0$): No steps taken; vacuously true.

Inductive step: Assume no heap allocation or atomic operation occurs in the first k steps. We show step $k+1$ also produces neither.

The only rule that allocates a heap location is Let-Shared. Let-Shared fires only when the declared type carries $+R$. We claim no subexpression of the k-step residual e_k has type $+R$ under any reachable environment.

Proof of claim: By the sharing discipline (Definition 6.3), a $-R$-typed expression can contain a $+R$-typed subexpression only if that subexpression was introduced by a Let-Shared elaboration, which requires an explicit $+R$ annotation in the source. Since $e : \tau^{-R,\mu}$ and the source contains no $+R$ annotation (by hypothesis), no subexpression of e has type $+R$. By Preservation (Lemma 7.7), types are maintained across reduction steps, so no subexpression of e_k has type $+R$ either.

Therefore Let-Shared never fires, no heap location is allocated, no reference count is created, and no atomic operation is performed. □ □

B.4 Full Proof of Race-Freedom

Full proof of Theorem 7.3. We model execution as an interleaved multithreaded system where each thread maintains its own stack σ_i and all threads share a common heap H. A data race occurs when two threads access the same memory location with at least one write, without intervening synchronization.

We case-split exhaustively on modifier combinations.

Case $-R$: Stack locations ℓ^{-R} are created by Let-Local and stored in the stack σ_i of the creating thread. By the sharing discipline, a $-R$ value cannot be stored in any $+R$ context (which would require Let-Shared). Therefore no other thread can obtain a reference to ℓ^{-R}. No other thread can access the location; no race is possible.

Case $+R\,-M$: Heap locations of type $\tau^{+R,-M}$ are immutable by the $-M$ modifier. The only applicable reduction rules read the location (Field-Read); no write rule applies to $-M$-typed objects. Concurrent reads are race-free by definition.

Case $+R\,+M$: Heap locations of type $\tau^{+R,+M}$ are proxy-wrapped (the `proxy` flag is set to `true` by Proxy-Wrap). The only rule that modifies such a location is Proxy-Dispatch. In the interleaved semantics, each Proxy-Dispatch step is an atomic transaction (the policy application P and the heap update are a single reduction step). Interleaving is between reduction steps, not within them. Therefore no two threads can simultaneously modify the object; accesses are serialized at the proxy boundary. No race is possible.

All cases covered; the program is race-free. □ □

B.5 Full Proof of Preservation

We give the key cases of the Preservation lemma omitted from the main text.

Case Field-Write-Local: The subject is $e.f_j := e'$ with type $\tau^{-R,+M}$. The reduction updates the stack location bound to e. By the typing rule, e has type $\tau^{-R,+M}$ and e' has type τ_{f_j}. After reduction, the modified stack location has the same object type with the updated field; the type is preserved.

Case Proxy-Dispatch: The subject is $\ell^{+R}\,[\text{dispatch}]\,(\overline{v})$ with type $\tau^{+R,+M}$. The policy P produces a new state of the same type (by policy-conformance). The heap location is updated with the new state; its type remains $\tau^{+R,+M}$.

B.6 Full Proof of Stack-Frame Confinement

Full proof of Lemma 6.4. Suppose $\Gamma \vdash v : \tau^{-R,\mu}$ and v is passed as argument x to $\mathbf{f}^{-R}(x{:}\tau^{-R,\mu}).\, e'$. We must show that no evaluation of $e'\{v/x\}$ stores v (or any alias thereof) in a $+R$-typed location.

Suppose for contradiction that at some reduction step, v is stored in a $+R$-typed binding via Let-Shared. This requires the source term to contain $\texttt{let}\ y{:}\tau^{+R,\mu'} = v\ \texttt{in}\ e''$. But such a term requires v to be ascribed type $\tau^{+R,\mu'}$, which differs from its declared type $\tau^{-R,\mu}$ in the ownership modifier. By the type rules, ownership modifiers are not implicitly coerced; the only legal transition from $-R$ to $+R$ is via an explicit Let-Shared with a $+R$ type annotation. Such an annotation is not present by hypothesis (the function parameter has type $\tau^{-R,\mu}$ and no $+R$ ascription appears in the body e').

Therefore no Let-Shared step fires for v or any alias, and v remains confined to the caller's stack frame throughout the call. Upon return, the callee's stack frame is popped and v's stack location becomes unreachable; no dangling reference exists. □ □