U Programming Language: Designed for Dual Pits of Success

A Multidimensional Modifier Algebra for Safe, Fast, and Injection-Free Systems Programming

ANONYMOUS AUTHOR(S)

Every annotation in U serves two masters simultaneously: it is a *safety invariant* enforced by the type system, and an *exact compiler oracle* that licenses a specific transformation. This *dual role* is the central design principle. $-R$ is not merely "stack-allocated"—it makes escape analysis unnecessary by construction. $-M$ is not merely "immutable"—it makes alias analysis unnecessary. $-N$ is not merely "non-null"—it makes null-flow analysis unnecessary. The zero-annotation configuration ($-R$, $-M$, $-N$) is simultaneously the safest path *and* the highest-performance path—a formal realisation of the *pit of success*: correct defaults require no annotation.

The annotation system is an eight-dimensional commutative algebra over three bipolar axes ($\pm R$, $\pm M$, $\pm N$) and five unipolar modifiers ($+V$, $+C$, $+A$, $+F$, $+W$). Commutativity is not incidental—it is the formal statement that modifier order never affects semantics, only the zero-annotation defaults determine what is implicitly applied. The **Modifier-Optimization Correspondence** (Theorem 1) establishes that every annotation is a *sound transformation license*: safety results follow as corollaries, not as separately proved properties.

From this foundation, three structural open problems are closed. **(1)** *Function coloring* [20]: the **Red-Blue Freedom** theorem proves $+A$ inference eliminates coloring; **Constant-Time Suspension** proves suspend/resume is $O(1)$ via continuation closures capturing exactly the live-across-suspension set, eliminating function coloring while preserving complete traces—no prior system eliminates coloring by compile-time inference while keeping $O(1)$ precise-frame suspension and complete traces together. **(2)** *Injection safety*: **Template Safety** proves that all nine compile-time template literals structurally prevent injection at the type level—not by sanitization, but because unescaped strings are type errors. **(3)** *i18n correctness*: **Localization Completeness** makes missing locale keys and arity mismatches compile errors.

Two further contributions exploit the dual-role principle. The async system (§5) makes the $+A$ dependency graph of a program identical to its execution DAG: $+A$ values are edges, coroutines are nodes, the scheduler follows readiness—topological ordering is implicit in the types, not separately declared. The compile-time domain (§10) is symmetric with the async domain: `z` and `a` are the same mechanism applied to two execution domains, giving compile-time metaprogramming, method replacement, and supply-chain security as consequences of one rule. The same principle yields *speed*: because the collection combinators are element-independent by construction, U auto-parallelizes them to SIMD or GPU without the dependency analysis a C compiler must perform (Proposition 6.2)—the fast path is the default path.

To our knowledge, U is the first language designed for two audiences simultaneously: human developers, and LLMs that autonomously write and reason about large codebases. The annotations that enforce correctness double as a machine-readable fact database per binding—ownership, mutability, nullability, async boundary, write policy—readable in $O(1)$ rather than reconstructed by analysis on every query. We prove 11 theorems in a formal calculus λ_U and compare U against Rust, Go, JavaScript, Swift, and Zig.

CCS Concepts: • **Software and its engineering** → **Type systems**; **Concurrent programming structures**; *Compilers*; *Memory management*.

Additional Key Words and Phrases: modifier algebra, ownership types, type systems, concurrency, memory safety, async programming, zero-cost abstractions

1 Introduction

1.1 Two Pits of Success

The phrase "pit of success" [1] describes a system where the easy path is the correct path. U embeds this property twice.[1] *Developer's pit:* the zero-annotation default ($-R\,-M\,-N$) is safest and fastest; every cost needs an explicit `+`; unsafe programs demand visible opt-ins. *Tooling pit:* every binding is a machine-readable declaration of ownership, mutability, null contract, async boundary, and write policy. Tools read these facts in $O(1)$ rather than reconstructing them per query. Infrastructure errors are structurally impossible; logic errors are targeted precisely because semantics are declared, not inferred.

The compiler's pit. A compiler processing U code with no optimizations still produces optimal code for the common case—the type annotations are a complete oracle for every major transformation: $-R$ replaces escape analysis; $-M$ replaces alias analysis; $+R\,+M$ replaces points-to analysis; $-R$ eliminates ARC; $+R\,-M$ eliminates synchronization. The same coincidence extends to *parallelism*: the natural way to write a bulk transformation—`.map`/`.filter`/`.on`—is element-independent by construction, so a single `+V` lowers it along one ladder—SIMD lanes, threads across the CPU's cores, or a GPU grid of warps (thousands wide) when the data is `+R(GPU)`—all with no loop-carried-dependence analysis. The ergonomic path is the parallel path: the developer reaches hand-tuned-C throughput by writing the obvious combinator, not by annotating for it—the pit of success expressed as speed. In typical languages these analyses are expensive, approximate, and interdependent. In U they are exact, free, and independent. We call this *compiler-transparent ownership*: here the annotation is the *optimization* license. U is fully typed: no `any`, no root `Object`, no escape hatch. The `:` type test is for narrowing only. Java, TypeScript, Go, and Swift each have escape hatches; U has none. If you cannot name the type, the design is incomplete.

1.2 Three Open Problems, One Mechanism

Two problems have resisted clean solutions across all major systems languages. (1) **Function coloring** [20]: async functions infect callers, splitting codebases into two incompatible worlds. U closes this by placing $+A$ on the *value*, not the function—no caller annotation required. (2) **Injection and i18n**: SQL/HTML injection and missing locale keys are runtime failures in every major language. U closes both through typed template literals that verify escaping and key presence at compile time.

1.3 The Modifier System

U's annotation system has three *bipolar* axes, each with a safe, cheap negative default and a capability-granting positive: $\pm R$ (ownership: stack vs. heap), $\pm M$ (mutability: immutable vs. mutable), and $\pm N$ (nullability: provably non-null vs. nullable). Five *unipolar* modifiers ($+V$, $+C$, $+A$, $+F$, $+W$) opt into additional capability domains. The algebra is commutative: modifier order never affects semantics. The zero-annotation path ($-R$, $-M$, $-N$) is the most capable: the compiler receives the maximum set of transformation licenses, producing the safest and fastest code.

This is the formal statement of "both pits of success simultaneously." The developer writes less (no annotation = correct default) and gets more (maximum optimization licenses). Adding a `+` always *costs* something—either performance headroom (+R allocates on the heap), or safety guarantees ($+M$ requires a write policy, $+N$ propagates through the type).

[1] A live language reference guide is available as supplementary material.

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.

The eight-combination modifier space and its cost model are summarized in Table 1.

1.4 Safety as a Corollary

The safety results—ARC-Freedom, Race-Freedom, Proxy Safety—follow as corollaries of the *same* optimization correspondence. When the compiler is licensed to eliminate ARC for $-R$ terms, use-after-free is structurally impossible: there is no reference count to go to zero. When it is licensed to omit synchronization for $-M$ terms, data races are structurally impossible: no concurrent write path exists. When $-N$ eliminates null-flow analysis, null dereference is impossible: `none` is not the zero value of non-nullable types, it does not exist in them.

This is not coincidence—it is the formal content of the **Modifier-Optimization Correspondence**: every annotation is simultaneously an invariant and a license, so the proof of optimization soundness *is* the proof of safety. Safety does not require a separate verification pass; it is a logical consequence of a compiler that respects its licenses. We regard this unification as the primary theoretical contribution.

1.5 Notation Correspondence

Formal notation ($\pm R$, $\pm M$, $\pm N$) is used in calculus and theorems; surface notation (`+R`, `+M`, etc.) in code. Table 2 gives the full correspondence.

1.6 Design Rationale for Modifiers

Both bipolar axes default to negation-of-capability ($-R$, $-M$, $-N$), so the zero-annotation path is cheapest and safest. Mutability is capability opt-in by design; ownership and nullability follow the same principle.

1.7 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 (Modifier annotations).

Formal	Surface	Meaning
$-R$ (default)	(no annotation)	stack-resident, no GC
$+R$	`+R`	heap, ARC; sub-policies: `+R(GPU)`, `+R(Network)`
$-M$ (default)	(no annotation)	immutable
$+M$	`+M`	mutable; sub-policies: `MVCC`, `Actor`, `Mutex`, `CRDT`
$-N$ (default)	(no annotation)	guaranteed non-null
$+N$	`+N`	nullable / may be `none`
`+V`	Vectorized	SIMD (numeric types)
`+C`	Cache domain	near-memory, prefetch-friendly
`+A`	Async domain	survives coroutine suspension; inferred from `a`
`+F`	Function value	stored callable
`+W`	Wield/stream	infinite or lazy; `I+W` from `[1..w]`

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	`a f`: may suspend; `a fetch(url)`: spawn a coroutine, `+A` inferred
`c`	copy	`c expr`: shallow copy; `c+R`, `c+R(GPU)`: copy to a region
`z`	compile-time	`z f func()`: compile-time fn; `z f __load__()`: module init
`d`	class definition	no struct/class split
`e`	error / throw	`e ErrType({...})`; `e.on(...)` to handle
`f`	named function	`f name(params): T`; `=>` for lambdas
`o`	import / capability	`o Mod`: import; `o Mod {caps}`: with capabilities; `o =>`: exports
`r`	return	`r => value`
`t`	this / self	`t.field`, `t.method()`
`w`	wield / ∞	`w value`: emit from generator; `[1..w]`: infinite range
`b`, `y`	reserved (freed)	`b` was break; `y` was yield—now `w value`
`none`	null value	multi-letter literal
`true`	boolean true	literal of type `L`
`false`	boolean false	literal of type `L`

Table 3. Complete U keyword set. All keywords are one letter except `none`, `true`, `false`—the three multi-letter built-in literals. Multi-letter requirement for identifiers prevents collisions.

```
// Local immutable (param default): no annotation needed
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 = load_config()
// Shared mutable: explicit +R +M, proxy-mediated
cache: Cache +R +M = Cache()
cache.put(key, value)  // dispatches through proxy
```

EXAMPLE 1.1 (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].on(transform)

// Shared/escaping closure: heap-allocated, refcounted
handler = f +R(event) = process_event(event)
button.on_click(handler)   // handler outlives this scope
```

1.8 Copy-on-Write, Classes, and Iteration

Shallow copy via the `c` prefix operator is the only mechanism for producing an independent $-R$ value from a $+R$ object, making promotion costs explicit. The target region rides the operator: `c+R` promotes a stack value to the heap, and `c+M(MVCC) local` copies a local into a shared MVCC-governed object (`c+R(GPU)` to device memory, bare `c` for the default). 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.

Idiomatic examples.

```
f compute_mean(data: [N]): N    // -R -M -N: zero cost default
    total = data.on((acc,val)=>acc+val, 0.0)
    r => total / data.length

cache: Cache +R +M = Cache() +M(MVCC)  // shared mutable
cache.set(key, val)                          // proxy-mediated

a f fetch(url: S): S +N                     // may suspend
pending: S +N +A = a fetch(url)             // opt-in: new coroutine
result:  S +N    = fetch(url)               // auto-await

page = html‘<h1>{title}</h1><p>{input}</p>‘ // injection = type error
gpu: [N32+V]+R(GPU) = c+R(GPU) cpu_data        // to GPU device memory
```

Every line above is the same algebra read in a different register. As C reduces a machine to bytes in memory and chess reduces a game to legal moves on a board, U reduces a program to values carrying modifiers through typed chains—and the results the rest of this paper proves are not separate features but readings of that one object: memory safety and zero-cost defaults (§3), the safety theorems as corollaries of one optimization correspondence (§4.1), coloring-free concurrency (§6), data-parallel speed by construction (§6), injection and localization safety (§7), and compile-time metaprogramming (§4). We flag each as it arrives with the same phrase—*the annotation is the license*—so the single principle stays visible as the fruit accumulates.

2 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.

2.1 Syntax

Definition 2.1 (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 2.2 (Terms).

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

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 $[\mathsf{dispatch}]$ annotation.

2.2 Values and Heaps

Definition 2.3 (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.*

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

2.3 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. ARC-Freedom (Theorem 4.2) follows directly.

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

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

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

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

Fig. 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.

2.4 Concurrent Operational Semantics

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

Definition 2.4 (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 2.5 (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:*

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

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

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 4.6). This is stated as:

Axiom 2.6 (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 the proxy policy table (supplemental).

3 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.

3.1 A Compilation Scheme

We define a compilation function $[\![\cdot]\!]$ mapping well-typed λ_U terms to a typed assembly language TAL_U [19]. The scheme translates modifier annotations directly into allocation and scheduling primitives—each annotation is a compiler oracle, not a hint.

Modifier-to-instruction mapping. The four principal cases are:

$$[\![e : \tau^{-R}]\!] = \mathsf{salloc}\ [\![\tau]\!]\ [\![e]\!] \quad \text{(stack allocation, no GC header)}$$
$$[\![e : \tau^{+R}]\!] = \mathsf{halloc}\ [\![\tau]\!]\ [\![e]\!] \quad \text{(heap allocation, ARC header)}$$
$$[\![e : \tau^{+A}]\!] = \mathsf{mkfiber}\ [\![e]\!] \quad \text{(coroutine; continuation closure capturing the } +A \text{ live set)}$$
$$[\![e : \tau^{+W}]\!] = \mathsf{mkcoro}\ [\![e]\!] \quad \text{(coroutine frame; wield emits one value)}$$

Stack frames for $-R$ values are reclaimed at scope exit without scanning. ARC for $+R$ values is emitted as inc/dec at use sites, elided when the borrow checker certifies no aliasing. Across a suspension, only the $+A$-typed live set is captured, into a heap-allocated continuation closure (Theorem 5.2); $-A$ values stay on the native stack and never enter the closure. $+A$ values are heap-resident and freely shareable. The type system and the runtime agree exactly: what the type marks $+A$ is what the closure captures—no whole-frame preservation, no more than the live-across-suspension set.

Wield and streams (`+W`).. The `+W` modifier marks a type as a lazy or infinite stream. `w value` inside a `T+W` function emits one value to the caller and continues—the U spelling of *yield*. `[1..w]` produces an `I+W` infinite range; `[0..w]*2` is the even numbers. `.map()` and `.filter()` propagate `+W`; `.take(n)` and `.first()` terminate it. `.reduce()`, `.sort()`, and `.last()` on `+W` are compile errors. No explicit `() =>` vs `_ =>` distinction: use `() =>` for no-param closures; single-character identifiers are reserved.

Async compilation. The scheduler calls resume when all antecedents are ready, realising topological ordering from data-dependency alone (no explicit graph).

Write-policy compilation. $+M(policy)$ annotations compile to proxy wrappers inserted at mutation sites:

- `MVCC`: writes go to a version log; reads see the last committed version.
- `Actor`: writes enqueue a message; a single serial executor drains.
- `Mutex`: writes acquire a spinlock or OS mutex by size heuristic.
- `CRDT`: writes call the registered merge function; no coordination.

The proxy is generated once per `+M` site and inlined; no vtable.

Template compilation. Each template tag of the form `t‘...‘` desugars to a TemplateNode constructor checked against the tag's type grammar. Raw S in a typed position (e.g. HTML attribute value) is a type error; safe types (`Templates.HTML`, etc.) pass through unchanged. The check is purely syntactic over the type lattice—no runtime escaping.

z f compilation. `z f` bodies are evaluated by the compiler itself at compile time, with no runtime caps available. `z f __load__()` produces a list of method-table patches; the linker applies them in `compile_order` order before emitting the binary. All patches are

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

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

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

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

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

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

Fig. 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.

verified against the full algebraic contract (signature, modifier algebra, error surface) before application; a mismatch is a compile error.

Theorem 3.1 (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.*

3.2 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.

3.3 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 3.2 (Modifier Subtyping). *The subtyping relation $<:$ on modifier pairs is the least relation satisfying:*

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

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)—standard covariance of immutability.

Remark 3.3 (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.*

3.4 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.

3.5 Well-Typedness and the Sharing Discipline

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

Definition 3.4 (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.

3.6 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 3.5 (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. 3.4), 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 3.6 (Scope of Lemma 3.5). *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. 3.4, clause 1) enforces this.*

4 Safety as Corollaries

4.1 The Modifier–Optimization Correspondence

Theorem 4.1 (Modifier–Optimization Correspondence). *For every well-typed closed expression $\emptyset \vdash e : \tau$, each modifier annotation licenses the indicated compilation transformation in $[\![\cdot]\!]$, and each transformation preserves observational equivalence:*

(1) *$-R$: every $-R$-typed subterm is compiled via* salloc*; no heap location or reference count is created for it.*
(2) *$-M$: no synchronization primitive is emitted for reads of a $-M$ object.*
(3) *$-N$: no null-check is emitted at a use of a $-N$ value.*
(4) *$+R+M$: every mutation compiles to a* dispatch *through the object's proxy; no direct store is emitted.*
(5) *$+A$: the value is registered as a dependency edge in the scheduler's readiness graph; its producer coroutine is a node.*

Each clause follows from the corresponding case of $[\![\cdot]\!]$ (§3) and Preservation (Lemma 4.9).

The zero-annotation path ($-R$, $-M$, $-N$) combines clauses (1)–(3): stack allocation, no synchronization, no null check—simultaneously safest and fastest. Safety corollaries follow.

The safety results below are corollaries of Theorem 4.1: each instantiates one or more clauses of the correspondence at a specific modifier and reads off the guarantee, so the burden rests on the correspondence itself.

4.2 ARC-Freedom

Theorem 4.2 (ARC-Freedom (Corollary of Theorem 4.1(1))). *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. Let-Shared is the only rule allocating a heap location, and it requires $+R$. Since $e : \tau^{-R,\mu}$, Let-Shared is never invoked—no heap locations, no refcounts, no biased-RC operations. The sharing discipline is preserved by Preservation (below). □ □

Corollary 4.3 (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.*

4.3 Race-Freedom

Definition 4.4 (Race). *A reachable configuration $\langle \sigma, H, \mathcal{T} \rangle$ has a* race *on ℓ if two distinct threads $e_i, e_j \in \mathcal{T}$ can in one step each access ℓ, at least one access is a write, and at least one access is* unmediated*—not a* Proxy-Dispatch *step.*

Note: MVCC and CRDT policies explicitly permit concurrent writers; they do not mutually exclude. The race-freedom property is therefore stated at the level of *mediation*, not serialization.

Theorem 4.5 (Mediation / Race-Freedom (Corollary of Theorem 4.1(2),(4))). *No reachable configuration of a well-typed λ_U program has a race.*

Proof. By cases on the type of the accessed location ℓ, using Preservation.
$-R$: Stack locations live in thread-private σ_i and never enter H (Definition 3.4.1); no two threads name the same ℓ^{-R}, so the race condition cannot hold.
$+R-M$: No write rule types a $-M$ object, so every access is a read; the race condition requires at least one write—impossible.
$+R+M$: The only typing rule that types a mutation of a $+R+M$ object is Proxy-Dispatch (Figure 2). By Preservation, a direct store on such an object is never typeable. Hence every write to ℓ^{+R+M} is mediated, and the unmediated-access clause is never satisfied. □ □

Each proxy policy delivers a separate consistency guarantee: Mutex gives mutual exclusion; Actor gives serial execution; MVCC gives snapshot isolation, permitting concurrent readers and writers; CRDT gives convergence, permitting concurrent writers with commutative merge. None of these require mutual exclusion, and MVCC/CRDT do not provide it—that is their purpose. Race-freedom (no *unmediated* access) is compatible with all four.

4.4 Proxy Safety

Definition 4.6 (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 4.7 (Proxy Safety (Corollary of Theorem 4.1(4))). *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. 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 4.8 (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.*

4.5 Type Soundness

We establish soundness via Progress and Preservation, by case analysis and induction over the rules of Figure 2; the standard substitution and canonical-forms lemmas and the full case enumeration are in the supplement.

Lemma 4.9 (Preservation). *If $\Gamma \vdash e : \tau$ and $\langle \sigma, H, e\rangle \longrightarrow \langle \sigma', H', e'\rangle$, then $\Gamma \vdash e' : \tau$.*

Proof. 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 4.10 (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. 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 4.11 (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 4.9 and 4.10 by standard subject-reduction argument. □ □

5 Surface Language

This section collects the surface-syntax decisions that make the modifier system usable but that the thesis does not depend on; a reader interested only in the formal contribution may skim it. Each choice follows one rule—every construct is an expression that yields a value—so the same grammar serves blocks, conditions, and pipelines.

Parentheses as block scope. U has no curly braces; parentheses `()` are the only grouping construct, used uniformly for argument lists, multi-line conditions, block expressions, and pipelines. Inside them newlines are free and the last expression is the value.

Magic methods. Six double-underscore methods form the v1 protocol (`__init__`, `__string__`, `__hash__`/`__equals__`, `__merge__` for `+M(CRDT)`), plus operator methods (`__add__` etc.) and three compile-time template hooks (`__check__`, `__validate__`, `__output__`).

Comprehensions. `{host, port}` uses variable names as keys; `{...obj, key: val}` spreads and overrides; `obj << {key: val}` atomically patches, recursing into inline fields and stopping at `+R` boundaries ($O(\log n)$ for large maps).

Namespaces and modules. `o Name` imports; `o Name => (...)` exports. `d Foo.Bar` is free namespacing. `t` absent from body ⇒ `+G`; present ⇒ instance.

Method overrides and explicit parent calls. `ParentClass.method(args)` calls a parent method (`t` implicit); no `super`. The compiler warns if a subclass `__init__` never calls its parent's. `t`-inference (absent ⇒ `+G`) is the one point where the compiler reads code content.

Type hierarchies, interfaces, and enums. U has no `enum` or `switch`—`d Color.Red : Color` declares a zero-field singleton subtype; `d Drawable!` declares an interface (all methods abstract); `f speak()!` marks one abstract method. Interface methods are never `+G`: interfaces constrain instances. Type-test dispatch (`expr : Type`) and typed `.on()` handlers replace switch; new variants need only a new handler—no existing code changes. `+G` marks class-level static fields.

Maps, generics, iteration, arity. Maps {`K: V`} preserve insertion order (language guarantee). Iteration passes `(value, key)`; handlers may declare fewer parameters—surplus args drop silently right-to-left. Maps expose `.keys()`, `.values()`, `.items()`. `ClassName({f:v})` creates instances (`new` absent); last parameter may be a typed default map; positional params have no defaults. Minimal generics: `d Stack[T]`, `d Tree[T]`; generic functions use interfaces.

1-based array indexing. Arrays are 1-indexed; `arr[0]` is a compile error. `arr[arr.length]` is the last element; `.last()` is sugar. `.slice(s?,e?)` is inclusive, 1-based.

Closures and arrow functions. `f name(params)` declarations hoist and do not capture `t`; arrows `(params) => expr` are anonymous values that do. Capture obeys the modifier rules: a $-R$ local in an escaping `+F +R` closure is a compile-time error. `->` is the type arrow, `=>` the value arrow.

Program as dependency graph. `+A` values are edges; coroutines are nodes. The scheduler follows readiness—it *is* Kahn's algorithm, not something that runs Kahn's algorithm on a separately-declared graph. The same code is simultaneously sequential code, a dataflow graph, and a build graph—the interpretation depends only on which values carry `+A`, which static tools can read off the annotations without any separate graph declaration.

Rejection throws. Auto-awaiting a rejected `+A` value throws into the same `e.on()` chain as every other error—no separate `.catch()`; one failure path.

`.on()` and array methods. `.on(fn)` replaces all bounded loops, passing `(value, index)` 1-based; the usual sequence methods (`.map`, `.filter`, `.reduce`, `.slice`, ...) follow. Both `.all(pred)` and `.any(pred)` are boolean predicates, while `.all()` and `.any()` on `[T+A]` are async join and race.

Suspension via continuation closures. JavaScript `async`/`await` loses call-stack frames across suspension points; Go copies entire stacks on growth. U takes the compile-time route: because every suspension point is statically known (a user-written `a` or an auto-await on a $+A$ value), a suspendable function is compiled—as V8 compiles JavaScript `await`—into a state machine whose *live-across-suspension set* is a heap-allocated continuation closure. This is a *stackless* coroutine transform (as in Rust `async` or C++20 coroutines): no per-coroutine call stack is retained. Suspending writes the resumption label into the closure and returns its handle to the scheduler—$O(1)$, no stack copy, no frame-chain inspection; resumption reloads the closure and jumps to the label. Crucially, the closure captures *exactly*

the values that cross the suspension boundary—the $+A$-annotated live set—and nothing more: $-A$ locals stay on the ordinary native (C/WASM) stack and never enter a closure, so their cost is zero. The captured set is thus precise rather than conservative: it holds only the referenced live-across-suspension values, not the whole activation record, which is what bounds the suspension frame to the statically-determined live set (Theorem 5.2). $-R$ immutable values may be shared into a closure without copying; a mutable value spilled across a suspension is exactly what $+A$ marks, so the $-R$ non-sharing invariant holds by construction. Thrown exceptions carry the chain of suspended closures for full-depth traces. Non-suspendable code uses the native call stack at zero cost.

Theorem 5.1 (Red-Blue Freedom). *No function's declared type depends on whether its callees suspend.*

Proof. Call sites are either auto-await ($x = g(\overline{e})$, giving $x : \tau$) or spawn ($x = \texttt{a}\, g(\overline{e})$, giving $x : \tau^{+A}$ inferred from `a`). The App rule reads only the callee's return type; suspendability is an internal compiler property on τ^{+F+A} values, invisible to App. The caller type-checks identically regardless. □ □

This is the ergonomic property a stackful runtime (Go) buys with whole-stack fibers—any call may suspend without changing its caller. U obtains it on a *stackless* substrate instead: because $+A$ inference makes every suspension point statically visible, the coloring never surfaces in types, yet each coroutine keeps only its precise live set (Theorem 5.2) rather than a whole stack. Coloring-freedom without stackfulness is the combination no prior system provides.

Theorem 5.2 (Constant-Time Suspension). *Suspending a coroutine at any suspension point takes $O(1)$ time and $O(1)$ additional space, independent of stack depth and number of live bindings. Resumption is $O(1)$. The continuation closure holds exactly the $+A$-annotated live-across-suspension set—no other frame is captured.*

Proof. By construction of $[\![\cdot]\!]$ on suspendable activations. Each suspension point compiles to a continuation closure whose captured environment is precisely the live-across-suspension set—the bindings the dataflow marks $+A$ (Section 5)—and whose code pointer is the resumption label. Yielding compiles to (i) a write of the resumption label into the closure and (ii) return of the closure handle (one reference) to the scheduler; neither operation inspects any caller frame, so both are $O(1)$ and copy nothing beyond the single handle. The captured set is fixed at compile time and does not grow with call depth or with the number of $-A$ locals (which remain on the native stack), so the additional space charged at suspension is $O(1)$ in the size of that statically-determined set. Resumption reloads the closure and jumps to the label, again $O(1)$. □

Automatic backpressure and concurrency policies. The default for `items.on(a handler)` is sequential: one item at a time, suspending at each inner `a` call. Concurrency is opt-in via a middleware policy prepended to the chain: `items.on(Concurrency.pool(10), a handler)` admits up to 10 concurrent coroutines; `Concurrency.drop_latest` drops when full. Each policy returns `none` (admit), `false` (drop this item), `true` (abort the pipeline), or suspends—the same three-signal convention as all `.on()` handlers.

6 Iteration, Hardware Oracles, and Arithmetic

The unipolar modifiers cash out here: `+V`/`+C` turn the collection combinators into a parallelization oracle (below), and `Q` extends the numeric tower. The conditional and arithmetic surface syntax is included for completeness.

Conditional operators. Four operators replace all conditional keywords (`if`, `else`, `switch`, `match`): `?!` (ternary: `?` = yes branch, `!` = else branch), `&` (guard: if a then b), `|` (fallback: first truthy), `??` (null coalesce). `?` and `:` sit at the condition line's indent, continuation lines are indented to stay in their branch, and the last expression in each branch is its value; chains nest without a delimiter.

```
grade = score >= 90 ? "A" : score >= 80 ? "B" : "F"   // chained else-if
```

THEOREM 6.1 (CONDITIONAL COMPLETENESS). *`?!/:/&/|/??` are expressively complete for all conditional patterns expressible with `if/else/switch/match`.*

PROOF. By exhibiting a desugaring of each construct into the operators. *if/else:* `if` c `then` a `else` $b \equiv c$ `?` a `:` b; a one-armed `if` c `then` $a \equiv c$ `&` a, whose value is `None` on the false branch. *switch/match on a scalar:* an n-arm dispatch on x desugars to a right-nested chain $x{=}v_1$ `?` e_1 `:` … `:` $e_{\texttt{default}}$. *match on a type* uses the `::` is-a test in the same chain (x `::` T_1 `?` …), total when the scrutinee's declared union is covered—the exhaustiveness the union-dispatch check already enforces. *fallback* chains (first non-`None`) are a `|` b `|` c. Since every arm maps to one of these forms and the operators nest without a block delimiter, any pattern expressible with the keywords is expressible with the operators. □ □

Iteration, ranges, and loop safety. `.on(fn)` replaces all bounded loops; ranges `[1..5]` (inclusive), `[1...5]` (exclusive), `[5..1]` (descending) feed it uniformly—compiler emits tight counter loops without allocation. Arrays are **1-indexed**: `arr.slice(s?,e?)` inclusive; `slice(-n)` = last n; `arr[0]` is a compile error. `w` requires reachable `b`; `w+W` declares an intentionally infinite loop (suppresses missing-`b` warning) and requires at least one `y` or `a` inside—no path may starve the scheduler. `T+W` is the yielding type: a stream that produces `T` values indefinitely; `T+W` absorbs `+A`.

Q: deferred rational arithmetic. `Q` replaces the decimal type with a rational pair (numerator, denominator) of *floating-point* components. Each is an ordinary IEEE 754 value, so `Q` represents small integer ratios *exactly* (integers below 2^{53} are exact in a double, so `0.1+0.2==0.3` holds structurally by cross-multiplication, no ε) while also admitting irrational or transcendental components as float approximations in either position. Crucially, `Q` is *symbolic by default*, exactly as `+A` is deferred: an expression nests—sums, products, ratios—without performing the division, and collapses only on assignment to a non-`Q` type (or when `=!` forces one IEEE division). Deferral buys precision: intermediates never round, so `Q(1,`10^{100}`)` survives arbitrarily deep computation where a float flushes to zero, removing vanishing gradients as a floating-point artifact. Bounded refinements fix precision or nesting depth (`Q2`: denominator ≤ 100 for exact currency; `Q`$[n,d]$: both). `I/I` returns `Q+N` (division by zero yields `none`, guarded before use); `I` and `U` extend to 2048 bits for cryptography.

Hardware domains, ML workloads, and numeric extensions. `+V` applies to any I, U, or N type; operations emit SIMD on CPU and GPU warps with `+R(GPU)`. On a function `+V` is a capability (vectorizable computation) licensed by $-E\ -\ D$—the purity a lockstep lane

requires is what an auto-vectorizer needs; on data it is vector-ready layout. The capability is earned by the constraint. `+C` adds prefetch hints. `+R(GPU)` places objects in GPU device memory; cross-boundary copies are explicit via `c+R(GPU)`. Location mismatch is a compile error.

Compiler-transparent parallelism. The combinators `.map`/`.filter`/`.on` carry their independence structurally, so U parallelizes them without the dependency analysis a C auto-vectorizer must perform (and conservatively fails) on a `for` loop.

PROPOSITION 6.2 (PARALLELIZATION WITHOUT ANALYSIS). *Let `xs.m(fn)` be `.map`, `.filter`, or `.on` over a collection whose element type is `-M` (or `-R`), with `fn` free of suspension and of writes to shared state. Then the n invocations of `fn` are data-independent, and any evaluation order—including full parallel execution across SIMD lanes (`+V`) or a GPU grid (`+R(GPU)`)—yields the same result as the sequential order. No loop-carried-dependence analysis is required to license the parallel lowering.*

PROOF. By the combinator's typing rule, `fn` is applied to one element and its result placed at that element's position; the rule gives `fn` no channel to another element's input or output. A write that could induce a cross-element dependence would be a mutation of shared state, which requires a `+M` (and `+R`) operand; the hypothesis excludes it, and the type system rejects any such write in `fn` statically. Hence the read/write sets of distinct invocations are disjoint, so they commute: the n applications form an independent family, and by confluence every evaluation order (sequential, SIMD-parallel, or grid-parallel) reaches the same normal form. The independence is therefore established from the types alone—no aliasing or loop-carried-dependence analysis of `fn`'s body is consulted. □ □

So the modifier is the parallelization license: a `+V` element type lowers directly to SIMD lanes, CPU threads, or—when `+R(GPU)`—a GPU grid of warps. `.reduce` is the one combinator carrying a cross-element dependence; it lowers to a parallel tree when its operator is associative and to a scalar fold otherwise. Default width follows the source (hardware width for `+V`, one in-flight for `+A`), tuned by `maxInFlight`.

Matrices, tensors, and the `@` operator. Nested arrays give arbitrary-rank tensors—a GPU matrix is `[[N32+V]]+R(GPU)`. `@` earns operator status—not a library call—because it licenses the compiler to inspect operand shape: small fixed dimensions unroll inline in a `+V` lane, large ones dispatch to a tuned BLAS (any rank as batched matmul, statically shape-checked). Scheduling shows the algebra again: a source's `+V`-ness selects `maxInFlight`'s regime—core count for a non-`+V` CPU pipeline, warp width for a `+V` one—so independent pipelines dispatch to CPU and GPU concurrently from the element type alone. `C64` and `C128` give complex numbers. *Generics (v1)*: class-level type parameters only, multi-letter by convention (`d Stack[ElemType]`); modifiers validate at instantiation (`[ElemType+V]` requires `ElemType` numeric). *Compile-time composition (`z f`)*: `z f` functions run at compile time, taking and returning typed function values. *Destructuring*: `{x,y}=point`; `[h,...t]=items`. Location mismatch is a compile error.

Arithmetic safety. Bounds, divide-by-zero, and overflow checks are on by default. `!` suffix opts out by the unsafe convention.

7 Context-Aware Templates and i18n

The same reading, now as *correctness*: a template tag is an annotation, and the annotation is the *escaping* license—an interpolation is well-typed only through its context's escaper.

Nine compile-time template tags cover all injection-prone sinks. `html` (S auto-escaped; `HTML` or `[HTML]` passed through; `[HTML]` auto-concatenates—the composition primitive for component trees); `sql` (S parameterized, `SQL` composes); `css`, `url`, `svg`, `md`, `command` follow the same per-type contract; `regex` (S is a compile error; only `Regex` sub-patterns; flags after closing backtick; invalid patterns are compile errors); `localized` (locale key and placeholder types verified at compile time; all locales bundled at build time, runtime selects with zero I/O). Raw `S` cannot reach any sink—the guarantee is structural.

Theorem 7.1 (Template Safety). *In all nine template tags, {**expr**} is context-safe: **S** is escaped, parameterized, or rejected; typed values compose; **[HTML]** auto-concatenates. Locale keys and placeholder types are verified at build time. Injection and missing-key crashes are compile-time errors.*

Proof. The tag is fixed at the call site (dynamic tags are rejected, so the sink context is statically known), and each interpolation $\{e\}$ elaborates to a call $\texttt{esc}_\tau(e)$ where τ is the tag's target grammar (HTML, SQL, ...) and $\texttt{esc}_\tau$ is that grammar's context escaper. The elaboration is driven by the typing rule for tagged literals: a template of tag τ types each hole against the escaper's domain, so a raw `S` in a sink position is *only* typeable through $\texttt{esc}_\tau$. There is no competing rule that admits a bare `S` into the concatenated result—the only way to bypass escaping is the explicit $\{\{e\}\}$ marker, which is a syntactically distinct form. Hence every well-typed interpolation is escaped or parameterized at the boundary, and an unescaped raw string in a structured context has no typing derivation: injection is a type error rather than a runtime condition. The argument is uniform across the nine tags because each supplies its own $\texttt{esc}_\tau$; only the escaper changes, not the rule. □ □

Theorem 7.2 (Localization Completeness). *In a well-typed λ_U program, every use of a **localized** template literal refers to a key that is present in the locale bundle with the correct argument arity. A missing key or an argument-count mismatch is a compile-time type error.*

Proof. The `z f __check__` hook receives key and argument list at compile time; the bundle is a finite map loaded at `z f __load__()`. Key lookup and arity check are decidable over finite, statically-known inputs; failure is a type error. □ □

Cooperative yield and declarative ordering. Three mechanisms yield a coroutine: (1) auto-await ($\tau^{+A} \to \tau$ on resolution); (2) `a` (immediate scheduler yield (via any `a` call)); (3) `a` n (timed yield, n ms lower bound). The event loop processes one run-queue entry per iteration (no macrotask/microtask split), eliminating JavaScript's microtask starvation. The $+A$ type graph is a statically visible dependency graph; `.on()` subscribers on one τ^{+A} form a typed pub/sub emitter, firing on resolution with zero additional API. Registration accrues (each `.on(fn)` appends a subscriber) and `.off(fn)` removes one by handler identity—the `.on`/`.off` pair of an event emitter, recovered from the type system rather than a bespoke class; where registrations are static, the subscriber set is a compile-time constant and dispatch stays direct.

8 Safety by Construction: Compile-Time Type Errors

Every error category in Table 4 is a compile-time type error in U. The program does not build. No runtime check, no linter, no style guide enforces these— the type system does.

We detail the four most consequential cases.

Error	How U prevents it	Runtime in
Unhandled error type	Declared `! ErrType` enables exhaustive checking	All languages
Non-boolean cond.	`?` condition and `&` left side require `L`; `5 ? 2 ! 8` is a type error	C, JS, Python
Use-after-free	$-R$ cannot be assigned to $+R$—type mismatch	C, C++, Zig
Null dereference	$-N$ default; $+N$ access without guard is type error	C, Java, JS, Go
Data race	$+R+M$ defaults to MVCC; unsafe sharing structurally impossible	Go, C, C++
Async violation	`a`/$+A$ co-required; mismatch is type error	JS, Rust async
Use-after-await	$+A$ annotation required on surviving locals	JS
Const mutation	$-M$ property refuses writes; binding $+M$ does not override	C++ (UB), JS
Closure scope escape	$+F+R$ cannot capture $-R$ locals—type error	C++ (UB), Swift
Injection (XSS/SQL/regex)	S structurally blocked from `HTML`/`SQL`/`Regex` sinks	All languages without typed templates
SQL injection	Raw $S \neq$ `SQL`; type error at call boundary	All languages
Surprise allocation	No implicit copies; static copy budget proven per function	C++, Swift
i18n missing key	`` localized`{key}` `` checks locale bundle at compile time	All languages
Zero index (`arr[0]`)	1-indexed arrays; `arr[0]` is a compile error	0-based languages
Accidental inf. loop	`[1..w].on(handler)` terminates when handler returns `true`; no exit path ⇒ compile error	All languages
Uncooperative loop	`w+W`: intentionally infinite; requires ≥ 1 `y` or `a` inside; starving = compile error	All languages

Table 4. Thirteen error categories eliminated by the U type system.

Non-boolean condition. The condition of `?!` and the left operand of `&` must be type `L`. In languages with implicit boolean coercion (C, JavaScript, Python), any truthy value silently passes a conditional test; in U, using an integer, string, or nullable where a boolean is required is a compile-time type error:

```
5 ? 2 ! 8          // ERROR: 5 is I, not L
count & process()  // ERROR: count is I, not L
user & show_panel() // ERROR: user is User +N, not L
count > 0 & process()  // OK: > returns L
L(count) ? 2 ! 8       // OK: explicit cast
count == 0 ? 2 ! 8     // OK: == returns L
```

`|` and `??` are exempt: they select between values, not test a condition.

Use-after-free. $-R$ (stack) values carry no refcount and cannot be shared. Assigning a $-R$ value to a $+R$ slot is a type error:

```
local: Vec2 = Vec2({1.0, 2.0})  // -R stack
ref: Vec2 +R = local            // ERROR: -R → +R
f make_point(): Vec2 +R
    pt: Vec2 = Vec2({0.0, 0.0}) // -R
    r => pt                     // ERROR: -R escapes scope
    r => c+R pt                  // OK: promote to heap first
```

There is no pointer arithmetic, no `free`, and no way to alias a stack address into a heap slot—use-after-free is structurally impossible.

Null dereference. The default modifier is $-N$: non-null, enforced. Every binding is non-null unless the programmer writes $+N$. Method calls and field accesses on $+N$ values are type errors without a prior null guard:

```
name: S +N = query()
len = name.length()     // ERROR: name is +N, may be none
name != none & len = name.length()  // OK: guarded
display = name ?? "---"   // OK: null-coalesced
```

Data race. Shared mutable state requires a declared proxy policy. $+R+M$ without write policy on $+M$ is a type error:

```
state: Config +R +M = Config()          // ERROR: +M needs write policy
state: Config +R +M +M(MVCC) = Config() // OK
```

There is no way to share a mutable object without declaring how concurrent writes are serialized or merged.

Async inference, ergonomics, and opt-in. `a` at the call site *infers* `+A` on the result—no annotation required, just as `x = 42` infers `I`. This makes concurrent code lightweight: `pending = a fetch(url)` is all that is needed. `a` works on *any* function—`a f` declaration means the function may internally suspend; `a` at the call site always creates the coroutine and always infers `+A`. Auto-lift: `a fn(pending)` where `pending : T+A` spawns a coroutine that awaits `p`, calls `f` with the resolved value, and returns `ReturnType+A`—even if `f` is not declared `a`. Multiple `+A` args are awaited concurrently (join semantics). Type annotations on `+A` are reserved for function signatures and documentation; call sites infer. The only error: `+A` on the LHS without `a`—no coroutine exists.

```
pending = a fetch(url)    // S+A inferred --- lightweight
pending: S+A = a fetch(url)  // explicit --- optional at call site
pending: S+A = fetch(url) // ERROR: +A without a coroutine
result = a combine(p1,p2) // auto-lift: awaits p1,p2, returns Result+A
f sync()                  // plain function
result = a sync()         // OK --- a upgrades any function, +A inferred
```

Promises, events, and types are one primitive. `T+A` is a typed future; the same dispatch that routes `e.on((err: T) => ...)` by error type routes resolution of any `T+A`—both are *typed arrival dispatch.* `a` at the call site *infers* `+A`; annotation is optional; `a` works on any function. `a fn(pending)` where `pending : T+A` auto-lifts: spawns a coroutine that awaits `p`, calls `f` with the resolved value, returns `ReturnType+A`—even if `f` was not declared `a`. **The program is isomorphic to its execution DAG**: `+A` values are edges, coroutines are

nodes, the scheduler follows readiness. Unlike spreadsheets, TensorFlow, or Dask, no second representation—no graph DSL, no decorator, no YAML—is required. The type annotation *is* the edge declaration.

Declaration rule. A function that *originates* a throw (`e ErrType(...)`) must declare `! ErrType` in its own signature. A function that merely *propagates* a callee's throw need not: the compiler infers the escaping surface automatically. This prevents proliferation—calling five functions with five different error types requires no re-declaration at each intermediate frame—while ensuring new throws are always a local, visible decision. Any function may optionally seal its surface with `! Type`; the compiler then verifies that exactly that type (and nothing else) escapes. For block-scoped catching, a `()` expression is the try-block: `e.on()` inside `()` applies only within that block.

9 Access Policies and the Proxy Model

All $\tau^{+R,+M}$ objects are proxy-mediated. The `+M(policy)` modifier declares how writes and reads are handled. It appears on the class definition (default for all instances) or at the binding site (per-instance override). Six policies: `Actor` (serialized queue), `MVCC` (readers never block), `CRDT` (commutative writes, merge-safe), `Mutex` (exclusive lock), `RAII` (cleanup fires on last-reference-drop), and `Network` (serializable across node boundaries).

Every `+R +M` binding requires a `+M(Policy)` declaration. The policy defines how concurrent reads, writes, and cleanup are handled. GC is one such policy—not the default.

Definition 9.1 (Policy Catalogue). *Write policies on `+M:` `+R(RAII)` (drop on last-ref), `+M(MVCC)` (optimistic versions), `+M(CRDT)` (commutative merge), `+M(Actor)` (queue), `+M(Mutex)` (exclusive lock), `+R(Network)` (remote proxy), `+R(GPU)` (GPU device memory).*

The default for `+R` is ARC without atomics: the single-threaded coroutine scheduler means refcounts need no atomic operations—cheaper than C++'s `shared_ptr`. For stack values (`-R`) the cost is zero: they disappear at scope exit.

```
d DbConn +R(RAII)     // +R(RAII): drop() fires on last ref
    f drop(conn: DbConn +R) => db.close(conn.handle)

f query(sql: S): S
    conn: DbConn +R = DbConn.open()   // refcount=1
    r => conn.exec(sql)               // conn refcount→0: drop() fires
    // No finally, no try-with-resources, no defer
```

Table 5 compares lifetime management across languages. Java and JavaScript use GC: non-deterministic, with pause costs and no structural cleanup guarantee. C++ uses manual allocation plus RAII destructors; `shared_ptr` provides ARC but with atomic overhead. Go uses GC with `defer` for explicit cleanup—forgettable. Rust enforces RAII via ownership; `Arc<Mutex<T>>` for shared mutable state. U declares the policy at the binding site: the choice is explicit, the compiler enforces it, and ARC is cheaper than in any other language that provides it.

Shared mutable state uses the same binding-site declaration—an MVCC policy lets readers hold a stable version while a writer installs the next:

```
config: AppConfig +R +M(MVCC) = AppConfig.load()
config.reload()  // new version; readers hold old version until done
```

Language	Default	Deterministic?	Shared mutable
Java	GC	No	`synchronized`, `volatile`
C++	Manual+RAII	Yes	`shared_ptr`, `mutex`
Go	GC	No	Channels, `sync.Mutex`
Rust	Ownership	Yes	`Arc<Mutex<T>>`
JavaScript	GC	No	Single-threaded only
U	`-R` free, `+R` ARC	Yes	`+M(Actor|MVCC|CRDT|Mutex)`

Table 5. Lifetime management and shared-mutable access across languages.

Composability. U's features compose because they share one model. The four pit-of-success properties: (1) default is safest ($-R\ -\ M\ -\ N$); (2) one mechanism for events, errors, and iteration (`.on()`); (3) error handling is a preamble, not a wrapper; (4) policy declared once at the binding site.

```
a f load_dashboard(ids: [I])
    e.on(MyLib.network_policies)         // typed policy, declared once
    pending: [Article +A] = ids.map((id) => a fetch_article(id))
    articles: [Article]   = pending.all()
    render(articles)
```

10 Compile-Time Metaprogramming

U's compile-time system is not a separate macro language bolted onto the runtime language—it is the same language in a different execution domain. Just as `a` marks the async domain (Section 5), the keyword `z` marks the compile-time domain. The symmetry is exact: `z f func()` declares a compile-time function exactly as `a f func()` declares an async function; within `z f`, the `z` keyword is required at call sites exactly as `a` is required within `a f`. `z f` functions accept typed function values alongside ordinary values, return transformed functions, and compose freely:

```
z f with_hooks(func: (S)→Data+A, before: ()→none,
               after: (Data)→none): (S)→Data+A
    r => z hooked_wrapper(func, before, after)   // z calls z

safe_get = z retry(
    z with_hooks(HttpClient.get, log_start, log_end),
    3, err => alert(err))    // regular fn passed freely
```

This mechanism connects cleanly to every other part of U. *Modifier algebra*: the contract check on method replacement uses the same seven-dimensional algebra, so replacing an `a f` method with a non-async function is a compile error, not a silent runtime mismatch. *Capability system*: `z f` functions run with zero runtime capabilities—the non-escalation rule that governs `+A` applies here, so a compile-time function cannot perform I/O; security is structural. *Template system*: the `z f __check__`, `z f __validate__`, and `z f __output__` hooks on template tags (Section 7) are themselves `z f` functions—template safety is metaprogramming applied to string-literal syntax. *Q arithmetic*: compile-time functions transform and partially evaluate symbolic `Q` expression trees before materialisation, enabling exact rational constant folding across module boundaries.

This unified design replaces—with a single mechanism—C preprocessor macros (textual, untyped, unhygienic), C++ template metaprogramming (a separate sublanguage with notoriously unreadable errors), Objective-C method swizzling (runtime, unsafe, no contract check), and Python decorators (runtime overhead on every call, no type safety).

Method replacement and call tables. Inside any `z f` (including the module initializer `z f __load__()`), any method on any class in scope may be replaced:

```
z f __load__()
    HttpClient.get = z retry(z trace(HttpClient.get, "http"), 3)
```

The compiler verifies the full algebraic contract: parameter types, return type, modifier algebra, and error surface must all match; violations are compile errors, not runtime surprises. Call-table entries are rewritten in-place at link time; all call sites resolve to direct jumps with zero dispatch overhead—superior to C++ vtables (two memory loads per call) and Objective-C message dispatch (runtime hash lookup). This enables AOP, mocking, and middleware injection with zero runtime cost and full type safety.

Just-in-time loading. Because `z f` runs in the compile-time domain, loading a file *is* compiling it: each module's `z f __load__()` is its own compile-time entry point, so a new file can be brought into a running program by compiling it just-in-time. Dynamic loading is thus a first-class operation with no separate safety story—the same contract and capability checks that gate static compilation gate a dynamically loaded module before its symbols link in, precisely because the loading happens in a compile-time context.

Module capabilities and traits. The `o` keyword ("outside") handles both import and export: it declares which namespace a file contributes to and what it may reach into. The first `o` statement names the module and its capability set, and compile-time modification capabilities take the `c` prefix—`o Mod { c HttpClient.get }` declares that `Mod` will replace `HttpClient.get` in its `__load__`. A *trait module* has an empty `o =>` {}: its whole effect is adding methods to existing classes (like Objective-C categories, but compile-time, contract-verified, and auditable). The security rests on one containment rule, applied uniformly: a module can reach into only what its own `o` declaration names, and a dependency it includes can reach into only what the *includer* was itself granted—capabilities flow strictly downward, never widening. A module that never declares `c Crypto.*` cannot gain it by pulling in a dependency that does, and cannot pass to a sub-module a capability it lacks. This is the same non-escalation rule that governs `+A` propagation, so the compile-time capability model, the async effect model, and the module system are one security discipline, not three.

Supply-chain security. Because the reachable capability surface is declared, not inferred, it can be audited and priced. `u.lock` records the SHA-256 hash and M-of-N signatures for every dependency, and the registry sets signing thresholds proportional to that surface: a module claiming `z Crypto.*` requires more independent reviewers than a no-cap module, precisely because the containment rule guarantees the declared surface is the *whole* surface—nothing it includes can widen it. A Merkle root hash over the full dependency tree serves as the program's immutable identity—the same hash used to pin LLM security audits to an exact, reproducible code state.

Limitations. The continuation-closure model does not support suspending *across* a C FFI boundary: FFI calls execute on the native C/WASM call stack and must return before the

enclosing coroutine can suspend, since a foreign frame has no compiler-generated continuation to capture. This is the one boundary the compile-time approach cannot cross; within U code, every suspension point is statically known and compiled to a closure that captures only the live-across-suspension set.

11 Related Work

Ownership and memory. Rust [14, 17] prevents aliased mutation via borrow checking; U instead *admits* shared mutation and disciplines it through the proxy algebra, allowing MVCC/CRDT policies inexpressible in safe Rust. Ownership types [8], regions [12, 27], and fractional permissions [6] also confine aliasing statically; U's $\pm R$ is coarser (heap/stack) but zero-annotation in the common case, with linear roots in [28].

Async programming. Go goroutines [26] solve coloring but copy full stacks; F# async [25] and JavaScript Promises [11] keep colored functions, the latter with microtask starvation. Swift actors [2] and Pony's deny capabilities [9] type-check actor isolation with dedicated constructs; U encodes actors as $\tau^{+R,+M}$ with a mailbox proxy policy, no new keyword.

Type-based safety. Template safety follows context-sensitive auto-sanitization [23], moved to compile time; the $+F$ modifier relates to higher-kinded types [18].

Effect systems. From polymorphic effects [16] through Koka [15] and Eff [3], effect systems generalize what U's $\pm E$/ $\pm D$ and `+A`/`+W` specialize: U trades a general handler algebra for two inferable bits and zero-annotation defaults.

Data-parallel and GPU languages. Futhark [13] also derives GPU parallelism from purity; DPJ [5] derives deterministic parallelism from an effect system—both support U's thesis that the *license* is the constraint, though each is a dedicated language while U's `+V` is one modifier in a general-purpose algebra. Halide [22] separates schedule from algorithm by hand; U's schedule is read off the types. Like PyTorch [21], U dispatches contractions to vendor BLAS rather than regenerating GEMM.

Linear and affine types. Linear Haskell [4] and Clean's uniqueness types enforce consume-exactly-once; U's $\pm R$ approximates the heap/stack split without full linearity.

Language design. Zig [29] shares explicit-allocation goals via allocator parameters where U uses modifiers. Gradual typing [24] informs U's inference-with-assertion posture; ATS [7] and capability calculi [10] inform the type-theoretic foundations, particularly the treatment of capabilities as type-level licenses.

12 Compositionality: When Abstractions Fall Out

The strongest claim U makes is not the presence of features but their *absence*: familiar abstractions emerge from combinations of the orthogonal primitives rather than being designed in. This places U in a lineage of systems each defined by a single unifying idea—Lisp reduces everything to lists, Unix to files, C to memory, Smalltalk to objects, Erlang to processes, the λ-calculus to functions, and relational algebra to relations; U's is *values flowing through typed chains under explicit dimensions*. Every *the annotation is the license* we flagged along the way was one reading of this same object; the abstractions below are the rest of the harvest. Table 6 shows abstractions that fall out of combining seven primitives (Values, Modifiers, Events, Chains, Policies, Coroutines, Domains), none requiring bespoke design.

Table 6. Combinatorial composition: abstractions that emerge from primitive combinations.

Combination	**What emerges**
Values × Chains	`map`, `filter`, `reduce`, pipelines
Values × Coroutines	Futures, promises
Coroutines × Chains	Streams, observables
Chains × Policies	Middleware, retries, backpressure
Events × Chains × Policies	Checked exceptions, supervision trees
Domains × Compile-Time	Typed templates—HTML, SQL, Cap'n Proto, EIP-712
Modifiers × Location	Ownership, distribution, GPU placement, CRDT
Values × Events × Coroutines × Chains × Policies	Erlang supervision, temporal workflows, orchestration

Table 6 is illustrative, not exhaustive: combinatorial richness from few primitives is *derivable*, not postulated.

13 Conclusion

To our knowledge, U is the first design to combine four properties prior languages provide only in part: coloring-free concurrency through compile-time $+A$ inference—the ergonomics Go achieves with stackful fibers, but with suspension frames that capture only the precise live-across-suspension set; ownership-based memory safety (as in Rust) while admitting multi-writer MVCC/CRDT policies safe Rust excludes; zero-analysis compilation from modifier oracles; and compile-time template and localization safety. We do not claim each property is individually new—each has precedent, cited above—but that their unification under a single modifier algebra, with safety results following as corollaries of the optimization correspondence, is what we have not seen elsewhere. The eight-dimensional modifier algebra $(\pm R, \pm M, \pm N, +V, +C, +A, +F, +W)$ makes each annotation simultaneously a safety invariant and a compiler oracle: three bipolar axes carry default-safe semantics without annotation, five unipolar modifiers opt into capabilities. Three structural open problems are closed by theorem—function coloring (Theorem 5.1), injection safety (Theorem 7.1), and i18n correctness—and beyond these the type system eliminates use-after-free, null dereference, data races, async violations, const mutation, and injection, each a compile-time type error rather than a runtime check.

The compile-time metaprogramming system (`z f`) handles AOP, mocking, middleware injection, just-in-time loading, and supply-chain security through the same domain-symmetric mechanism as async; deferred rational arithmetic (`Q`) with `=!` eliminates float instability in scientific and ML workloads.

On LLMs and the need for new languages. Knowing the right pattern and structurally enforcing it differ: an LLM can recommend MVCC but cannot prevent the next developer from omitting the write policy; static checking can. Declared annotations cost $O(1)$ to read, whereas re-inferring structure scales with codebase size—so the annotations that enforce correctness for human developers double as externalized memory for the coding agents that reason across sessions.

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] Apple Inc. 2023. Swift Concurrency: Actors. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency/.

[3] Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 5:1–5:29. doi:10.1145/3158093

[5] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*. 97–116.

[6] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS)*. 55–72.

[7] 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.

[8] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. 48–64.

[9] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *AGERE!* 1–12.

[10] 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.

[11] ECMA International. 2024. ECMAScript 2024 Language Specification. https://tc39.es/ecma262/.

[12] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *PLDI*. 282–293.

[13] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates. In *PLDI*. 556–571.

[14] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34.

[15] Daan Leijen. 2014. Koka: Programming with Row-Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming (MSFP@ETAPS 2014) (EPTCS, Vol. 153)*. 100–126.

[16] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. 47–57.

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

[18] Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. In *OOPSLA*.

[19] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems* 21, 3 (1999), 527–568.

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

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI*. 519–530.

[23] Joel Samuel, Prateek Saxena, and Dawn Song. 2011. Context-Sensitive Auto-Sanitization in Web Templating Languages. In *CCS*.

[24] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*.

[25] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# Asynchronous Programming Model. In *PADL*. 175–189.

[26] The Go Authors. 2024. The Go Programming Language Specification. https://go.dev/ref/spec.

[27] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176.
[28] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. 347–359.
[29] Zig Software Foundation. 2024. Zig Programming Language Reference. https://ziglang.org/documentation/master/.