Mutation Is Local and Explicit:

Controlled Effects and Linear Ownership
in the JAPL Programming Language

Matthew Long
The JAPL Research Collaboration
YonedaAI Research Collective
Chicago, IL
`matthew@yonedaai.com`

March 2026

Abstract

Shared mutable state is the single most prolific source of defects in concurrent software. We present JAPL's approach to this problem: a *dual-layer* type system that separates *pure values*—immutable, garbage-collected, and freely shared—from *linear resources*—mutable, ownership-tracked, and deterministically released. The pure layer handles the overwhelming majority of application logic with no annotation burden, while the resource layer gives systems-level control over files, sockets, buffers, and foreign memory through an ownership discipline rooted in linear type theory and Girard's linear logic. We formalize the dual-layer system, prove type safety (progress and preservation), and show that well-typed JAPL programs satisfy *resource safety*: every acquired resource is released exactly once. A detailed comparison with Rust's borrow checker, Haskell's `IO` monad, Clean's uniqueness types, and several other approaches demonstrates that JAPL's design achieves a favorable trade-off between annotation burden, safety guarantees, and runtime performance.

Keywords: linear types, ownership, mutation, resource safety, effect systems, substructural type systems, dual-layer runtime

Contents

1 Introduction 3
1.1 The Mutation Problem 3
1.2 Contributions 4
1.3 Paper Organization 4

2 Background and Related Work 4
2.1 Substructural Type Systems 4
2.2 Linear Logic 5
2.3 Rust's Ownership and Borrowing 5
2.4 Clean's Uniqueness Types 5
2.5 Linear Haskell 6
2.6 Region-Based Memory Management 6
2.7 Capability-Based Security 6
2.8 Austral, Roc, and Koka 6
2.9 Algebraic Effects and Handlers 7

3 Formal Framework 7
3.1 Linear Type Theory 7
3.2 Connection to Girard's Linear Logic 8
3.3 Categorical Semantics 8
3.4 Compact Closed Categories and Duality 9

4 JAPL's Dual-Layer Model 9
4.1 Design Philosophy 9
4.2 The Pure Layer 9
4.3 The Resource Layer 10
4.4 The Split in Practice 11
4.5 Formal Typing Judgment 12

5 Ownership Semantics 12
5.1 Owned$\langle T\rangle$ 12
5.2 Consumption and Return Patterns 12
5.3 Borrowing 13
5.4 Region-Based Inference 14
5.5 Uniqueness Inference 14

6 Resource Types 15
6.1 Declaring Resource Types 15
6.2 Resource Lifecycle 15
6.3 The `use` Binding 16
6.4 Parametric Resource Types 16
6.5 Freezing: Resource to Value Conversion 16

7 Interaction Between Layers **17**
7.1 Pure Functions Using Resources . . . 17
7.2 Ownership Transfer Between Processes . . . 17
7.3 The Escape Hatch . . . 18
7.4 Effect System Interaction . . . 18

8 Comparison with Related Languages **19**
8.1 Overview . . . 19
8.2 Rust: Full Borrow Checker . . . 19
8.3 Go: GC Everything . . . 19
8.4 Haskell: The IO Monad . . . 20
8.5 Erlang: Copy Everything . . . 20
8.6 OCaml: Mutable Refs . . . 20
8.7 Clean: Uniqueness Types . . . 20
8.8 Summary of Trade-offs . . . 21

9 Implementation **21**
9.1 Architecture Overview . . . 21
9.2 Type Checker Implementation . . . 22
9.3 Linearity Checking Algorithm . . . 22
9.4 Runtime Memory Model . . . 22
9.5 Compile-Time Ownership Verification . . . 23
9.6 Performance Characteristics . . . 23

10 Formal Proofs **24**
10.1 Syntax of λ^{JR} . . . 24
10.2 Operational Semantics . . . 24
10.3 Type Safety . . . 25
10.4 Resource Safety . . . 26

11 Discussion **27**
11.1 When Ownership Pressure Leaks . . . 27
11.2 Mitigations . . . 27
11.3 The 80/20 of Resource Management . . . 28
11.4 Effect System Synergy . . . 28
11.5 Comparison with Monadic IO . . . 28
11.6 Limitations . . . 29
11.7 Future Work . . . 29

12 Conclusion **29**

A Complete Typing Rules for λ^{JR} **32**

B Extended Examples **33**
B.1 Database Connection Pool . . . 33
B.2 State Effect with Linear Resources . . . 34
B.3 Safe FFI Wrapping . . . 34

1 Introduction

The history of programming language design can be read, in large part, as a long negotiation with mutable state. Assembly gives the programmer direct access to every register and memory cell. C organizes that access into functions and structs but leaves aliasing entirely to the programmer's discipline. Java introduces garbage collection and access modifiers but permits unrestricted sharing of mutable objects across threads. Haskell confines all side effects to the `IO` monad, achieving referential transparency at the cost of pervasive monadic plumbing. Rust enforces ownership and borrowing rules that prevent data races at compile time, but its borrow checker pervades *every* layer of the program, even pure data transformations that could never cause a data race.

Japl takes a different position: **mutation is local and explicit**. The language is organized around two sharply delineated layers:

(i) A **pure layer** where values are immutable, garbage-collected, and freely shared. Most application code—business logic, data transformations, protocol handling—lives here.

(ii) A **resource layer** where mutable resources (file handles, network sockets, GPU buffers, FFI pointers) are tracked by a linear ownership discipline. Resources must be consumed exactly once: used and then released.

This separation is the second of Japl's seven core design principles[1] and is the subject of this paper.

1.1 The Mutation Problem

Shared mutable state creates three interrelated difficulties:

Concurrency bugs. When two threads can both read and write the same memory location, the result depends on scheduling—a data race. Languages without mutation restrictions must rely on locks, atomic operations, or transactional memory, each introducing its own class of bugs (deadlocks, priority inversion, live-lock).

Aliasing and invalidation. When multiple references point to the same mutable object, a mutation through one reference can invalidate assumptions made through another. The classic example is iterator invalidation: removing an element from a collection while iterating over it. More subtle is the *borrowed reference* problem in systems code: a pointer into a buffer becomes dangling after the buffer is reallocated.

Reasoning difficulty. Equational reasoning—substituting equals for equals—fails in the presence of mutation. If `f(x)` mutates `x`, then `let y = f(x); g(y, x)` is not equivalent to `g(f(x), x)`, because the second occurrence of `x` observes different state. This makes refactoring hazardous and formal verification prohibitively expensive.

Japl's dual-layer design addresses all three problems. Pure values cannot be mutated, so they are immune to data races, aliasing bugs, and reasoning failures. Resources *can* be mutated, but ownership tracking ensures that at most one reference exists at any time, restoring the ability to reason locally about mutation effects.

[1]The seven principles are: (1) Values are primary, (2) Mutation is local and explicit, (3) Concurrency is process-based, (4) Failures are normal and typed, (5) Distribution is a native concern, (6) Functions are the unit of composition, (7) Runtime simplicity matters as much as type power.

1.2 Contributions

This paper makes the following contributions:

1. We present a formal account of Japl's dual-layer type system, connecting it to Girard's linear logic and the theory of substructural type systems (Sections 3 and 4).
2. We define the ownership semantics for the resource layer, including consumption, borrowing, region-based inference, and uniqueness inference (Section 5).
3. We formalize Japl's resource types and prove that well-typed programs satisfy *resource safety*: every acquired resource is eventually released exactly once (Sections 6 and 10).
4. We describe the interaction between the two layers, including the borrowing protocol and escape hatches (Section 7).
5. We provide a detailed comparison with six other language designs (Section 8) and describe an implementation strategy for the dual-layer runtime (Section 9).

1.3 Paper Organization

Section 2 surveys related work. Section 3 develops the formal framework: linear logic, substructural type systems, and categorical semantics. Section 4 presents Japl's dual-layer model. Section 5 details ownership semantics. Section 6 defines resource types. Section 7 describes layer interaction. Section 8 compares with related languages. Section 9 discusses implementation. Section 10 contains formal proofs. Section 11 reflects on practical trade-offs. Section 12 concludes.

2 Background and Related Work

2.1 Substructural Type Systems

The structural rules of a type system determine how hypotheses in a typing context may be used. Classical logic (and the simply-typed lambda calculus) admits three structural rules: *weakening* (a hypothesis may be ignored), *contraction* (a hypothesis may be duplicated), and *exchange* (hypotheses may be reordered). Dropping one or more of these rules yields a *substructural* type system [5]:

Definition 2.1 (Substructural type systems)**.** *Let Γ denote a typing context. The four principal substructural disciplines are:*

(a) ***Linear****: No weakening, no contraction. Every variable must be used exactly once.*

(b) ***Affine****: No contraction, but weakening is allowed. Every variable may be used at most once.*

(c) ***Relevant****: No weakening, but contraction is allowed. Every variable must be used at least once.*

(d) ***Ordered****: No weakening, no contraction, no exchange. Variables must be used exactly once, in order of introduction.*

Japl's resource layer employs *linear* types: each resource must be consumed exactly once. The pure layer uses standard structural types (with all three rules), since immutable values can be freely shared, duplicated, or ignored.

2.2 Linear Logic

Girard's linear logic [1] provides the proof-theoretic foundation for linear type systems. The key connectives are:

- $A \multimap B$ (linear implication): consuming one A produces one B.
- $A \otimes B$ (tensor / multiplicative conjunction): both A and B are available simultaneously, each used exactly once.
- $!A$ (of course / exponential): an unlimited supply of A, recoverable as many times as needed.
- $A \,\&\, B$ (with / additive conjunction): a choice between A and B, where only one is consumed.
- $A \oplus B$ (plus / additive disjunction): exactly one of A or B is provided.

The exponential $!A$ is critical: it bridges linear and unrestricted worlds. In Japl, the pure layer corresponds to the !-fragment of linear logic: every pure value implicitly carries an exponential modality, meaning it can be freely duplicated and discarded. The resource layer corresponds to the linear fragment proper.

2.3 Rust's Ownership and Borrowing

Rust [11] pioneered the use of ownership and borrowing as a practical mechanism for memory safety without garbage collection. Rust's key rules are:

1. Each value has a single owner.
2. Ownership can be transferred (moved).
3. References can be borrowed: either one mutable reference *or* any number of immutable references, but not both simultaneously.
4. Values are dropped (destructors run) when the owner goes out of scope.

The RustBelt project [12] formalized Rust's type system using the Iris separation logic framework, proving that well-typed Rust programs (even those using `unsafe`) satisfy memory safety when the unsafe blocks are locally sound.

Japl differs from Rust in a fundamental architectural decision: Rust applies ownership to *all* values, requiring lifetime annotations even for purely functional data transformations. Japl restricts ownership to the resource layer, leaving the pure layer free of ownership concerns.

2.4 Clean's Uniqueness Types

Clean [13, 14] uses *uniqueness types* to ensure that a value has at most one reference. If a value is unique, it can be destructively updated in place without violating referential transparency, because no other reference can observe the mutation.

Clean's uniqueness attribute `*` is propagated by the type checker. For example, `*File` denotes a unique file handle that can be read or written. Uniqueness types are closely related to linear types: a unique value must be used exactly once (or threaded through a computation that returns a new unique value).

Japl's resource layer is inspired by Clean's approach but extends it with explicit `own` and `ref` qualifiers, region inference, and integration with an algebraic effect system.

2.5 Linear Haskell

Linear Haskell [10] extends Haskell with linear arrow types $a \multimap b$, allowing functions to specify that an argument must be consumed exactly once. The key insight is *retrofitting*: linear types are added to an existing language without breaking backward compatibility, by making linearity a property of function arrows rather than types.

JAPL takes a complementary approach. Rather than adding linearity to arrows, JAPL adds linearity to a distinguished class of *types* (resource types). This makes the boundary between linear and unrestricted code syntactically visible.

2.6 Region-Based Memory Management

Tofte and Talpin [15] introduced region-based memory management, where objects are allocated into lexically scoped regions and deallocated en masse when the region goes out of scope. The Cyclone language [16, 17] brought regions to a C-like setting with static safety guarantees.

Region inference automatically determines which region each allocation belongs to, minimizing annotation burden. JAPL uses region-like reasoning in its ownership inference pass: the compiler determines the lifetime of each resource and inserts release operations at the appropriate scope boundaries.

2.7 Capability-Based Security

The capability model of security [18, 19] organizes access control around unforgeable tokens (capabilities) rather than identity-based access control lists. A process can only perform an operation if it holds the appropriate capability.

JAPL applies this principle to resource access: a file can only be read if the function holds an `Owned<File>` or `ref File` capability. The effect system (`Io`, `Net`) further restricts which capabilities can be exercised in a given context.

2.8 Austral, Roc, and Koka

Several recent languages explore design points closely related to JAPL's.

Austral. Austral [37] is a systems language with linear types in which every type is either *free* (unrestricted, may be copied and discarded) or *linear* (must be consumed exactly once). The dichotomy mirrors JAPL's pure/resource split, but Austral applies it within a single layer: there is no garbage-collected pure heap, and all memory management is explicit. JAPL's dual-layer design adds a GC-managed pure layer for ergonomics, and borrowing via `ref` T for non-consuming reads, which Austral lacks.

Roc. Roc is a functional language that targets application-level programming and uses a capability-based effect system in which all side effects flow through *platforms*. Like JAPL, Roc keeps the vast majority of code pure and pushes effects to the boundary. Unlike JAPL, Roc does not expose linear or ownership types to the programmer; resource management is delegated entirely to the platform layer.

Koka. Koka [30] pioneered row-typed algebraic effects with automatic effect inference. JAPL's effect system draws on Koka's design, but Koka does not provide linear types or an ownership discipline for resources. The combination of row-typed effects *and* linear resource tracking is a distinctive feature of Japl's design.

2.9 Algebraic Effects and Handlers

Algebraic effects [20, 21] provide a structured alternative to monads for describing side effects. An effect signature declares a set of operations; an effect handler provides implementations for those operations.

Japl's effect system (`Io`, `State[s]`, `Fail[e]`, etc.) draws on algebraic effects. Effects form a commutative monoid under composition, and effect handlers serve as natural transformations from effectful computations to pure results [22]. The interaction between effects and linear types is a key contribution of this paper: resource operations are effectful (`with Io`), and the linear discipline ensures that effect handlers cannot accidentally duplicate or discard resources.

3 Formal Framework

This section develops the type-theoretic and logical foundations of Japl's dual-layer system.

3.1 Linear Type Theory

We begin with a standard presentation of the linear lambda calculus [2, 3].

Definition 3.1 (Linear types). *The types of the linear lambda calculus are given by the grammar:*

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid {!A} \mid A \,\&\, B \mid A \oplus B \mid \mathbf{1} \mid \mathbf{0}$$

where α ranges over type variables, $\multimap$ is linear function space, $\otimes$ is multiplicative conjunction, $!$ is the exponential, $\&$ is additive conjunction, $\oplus$ is additive disjunction, $\mathbf{1}$ is the unit of $\otimes$, and $\mathbf{0}$ is the empty type.

Definition 3.2 (Linear typing context). *A* linear context Δ *is a multiset of typing assumptions $x : A$ where each variable x appears at most once. An* unrestricted context Γ *contains assumptions $x : A$ where x may be used arbitrarily many times. A* mixed context *is a pair $\Gamma; \Delta$.*

The typing judgment $\Gamma; \Delta \vdash e : A$ asserts that expression e has type A under unrestricted context Γ and linear context Δ, where *every* assumption in Δ is consumed exactly once.

Definition 3.3 (Core typing rules). *The following rules govern the linear lambda calculus with exponentials:*

$$\frac{}{\Gamma; x : A \vdash x : A}\ \textsc{Var-Lin} \qquad \frac{x : A \in \Gamma}{\Gamma; \cdot \vdash x : A}\ \textsc{Var-Un} \qquad \frac{\Gamma; \Delta, x : A \vdash e : B}{\Gamma; \Delta \vdash \lambda x.\, e : A \multimap B}\ {\multimap}\text{-}I$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : A \multimap B \qquad \Gamma; \Delta_2 \vdash e_2 : A}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\ e_2 : B}\ {\multimap}\text{-}E \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : A \qquad \Gamma; \Delta_2 \vdash e_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2) : A \otimes B}\ \otimes\text{-}I$$

$$\frac{\Gamma; \Delta_1 \vdash e : A \otimes B \qquad \Gamma; \Delta_2, x : A, y : B \vdash e' : C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\ (x, y) = e\ \mathbf{in}\ e' : C}\ \otimes\text{-}E \qquad \frac{\Gamma; \cdot \vdash e : A}{\Gamma; \cdot \vdash {!e} : {!A}}\ !\text{-}I$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : {!A} \qquad \Gamma, x : A; \Delta_2 \vdash e_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\ {!x} = e_1\ \mathbf{in}\ e_2 : B}\ !\text{-}E$$

The !-I rule requires that the expression e use *no* linear resources: only unrestricted values can be promoted to the exponential. This is the formal counterpart of JAPL's rule that pure values cannot capture resources.

3.2 Connection to Girard's Linear Logic

The Curry-Howard correspondence extends to linear logic: proofs correspond to programs, propositions to types, and cut-elimination to evaluation [1, 4].

Proposition 3.4 (Curry-Howard for linear logic)**.** *Under the linear Curry-Howard correspondence:*

(i) $A \multimap B$ *corresponds to a function consuming one A to produce one B.*

(ii) $A \otimes B$ *corresponds to a pair where both components must be used.*

(iii) $!A$ *corresponds to a value that may be duplicated or discarded.*

(iv) $A \,\&\, B$ *corresponds to a lazy pair (choose one projection).*

(v) $A \oplus B$ *corresponds to a tagged union (one alternative is provided).*

In JAPL's dual-layer system, the Curry-Howard correspondence manifests directly: pure-layer types correspond to the !-fragment (all types implicitly carry !), while resource-layer types correspond to the linear fragment (no implicit !).

3.3 Categorical Semantics

The categorical semantics of linear types provides a rigorous foundation for the dual-layer design.

Definition 3.5 (Symmetric monoidal closed category)**.** *A* symmetric monoidal closed category *(SMCC) $(\mathcal{C}, \otimes, I, \multimap)$ consists of a category $\mathcal{C}$ equipped with a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, a unit object I, and an internal hom $\multimap$ such that $- \otimes A \dashv A \multimap -$ for every object A.*

Definition 3.6 (Linear-non-linear model)**.** *A* linear-non-linear (LNL) model *[8] consists of:*

(i) A Cartesian closed category $\mathcal{C}$ (the "non-linear" or "intuitionistic" part).

(ii) A symmetric monoidal closed category $\mathcal{L}$ (the "linear" part).

(iii) A symmetric monoidal adjunction $F \dashv G : \mathcal{L} \to \mathcal{C}$ where $F : \mathcal{C} \to \mathcal{L}$ is a symmetric monoidal functor.

The exponential $!$ is recovered as the comonad FG on $\mathcal{L}$.

Theorem 3.7 (Benton's LNL correspondence [8])**.** *The category of LNL models is equivalent to the category of models of intuitionistic linear logic (ILL) with an exponential $!$ satisfying the standard structural rules (weakening, contraction, dereliction, promotion).*

This result is directly relevant to JAPL:

Remark 3.8 (JAPL as an LNL model)**.** *JAPL's dual-layer system instantiates the LNL framework:*

- *$\mathcal{C}$ is the category of pure JAPL types, which is Cartesian closed (it has product types, function types, and a terminal object `Unit`).*

- *$\mathcal{L}$ is the category of resource* Japl *types, which is symmetric monoidal closed (tensor is pair-of-resources, linear function is ownership-transferring function).*
- *The functor $F : \mathcal{C} \to \mathcal{L}$ embeds pure values into the resource layer (a pure value can always be used where a resource is expected, since it carries an implicit $!$).*
- *The functor $G : \mathcal{L} \to \mathcal{C}$ "forgets" linearity—this corresponds to the `freeze` or `snapshot` operations that convert a resource to an immutable value.*

3.4 Compact Closed Categories and Duality

In the full linear logic setting, the multiplicative fragment admits a notion of *duality*: every type A has a dual $A^{\perp}$, and the category of types forms a $*$-autonomous category [9].

Definition 3.9 ($*$-Autonomous category). *A* $*$-autonomous category *is a symmetric monoidal closed category $(\mathcal{C}, \otimes, I, \multimap)$ equipped with a dualizing object $\perp$ such that the canonical morphism $A \to (A \multimap \perp) \multimap \perp$ is an isomorphism for every object A.*

While Japl does not expose full classical linear logic to the programmer, the duality principle manifests in the resource layer's *consumption pattern*: acquiring a resource of type R creates an obligation (dual) to release it. The linear type system ensures that this obligation is always discharged.

4 JAPL's Dual-Layer Model

This section presents the core design of Japl's two-layer type and memory system.

4.1 Design Philosophy

The dual-layer model is motivated by an empirical observation: **most code does not need mutation**. Business logic, data transformations, protocol parsing, serialization, routing—all of these are naturally expressed as pure functions over immutable values. Mutation is needed only at the boundaries: reading files, writing sockets, managing GPU memory, interacting with foreign libraries.

By restricting ownership tracking to the resource layer, Japl achieves two goals simultaneously:

1. **Ergonomics**: The vast majority of code is free of ownership annotations, lifetime parameters, and borrow-checker errors.
2. **Safety**: The code that *does* interact with mutable resources is subject to a rigorous linear discipline that prevents resource leaks, use-after-free, and double-free.

4.2 The Pure Layer

The pure layer encompasses all immutable values: algebraic data types, records, tuples, lists, maps, strings, closures, and numeric types.

Definition 4.1 (Pure types). *The pure types of* Japl *are given by:*

$$\tau ::= \alpha \mid \texttt{Int} \mid \texttt{Float} \mid \texttt{Bool} \mid \texttt{String} \mid \texttt{Unit} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \texttt{List}[\tau] \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n \mid \rho\}$$

where α ranges over type variables and ρ ranges over row variables.

Pure values have the following properties:

- **Immutable**: Once created, a pure value never changes.
- **GC-managed**: Memory is reclaimed by a generational, per-process garbage collector.
- **Freely shareable**: Pure values can be passed to any number of functions, stored in any number of data structures, and sent to any number of processes.
- **Structurally typed**: Two values with the same structure have the same type (row polymorphism for records).

```
-- Pure values: immutable, GC-managed, freely shareable
let data = [1, 2, 3, 4, 5]
let copy = data -- sharing is fine; data is immutable

-- Pure function: transforms data, no side effects
fn transform(data: List[Int]) →List[Int] =
  List.map(data, fn x →x * 2)

-- Composition of pure functions
fn process_order(order: Order) →Invoice =
  order
  ▷ validate
  ▷ calculate_totals
  ▷ apply_discounts
  ▷ generate_invoice
```

Listing 1: Pure-layer code: no ownership concerns

4.3 The Resource Layer

The resource layer encompasses all mutable, owned values: file handles, network sockets, database connections, GPU buffers, FFI pointers, and mutable arrays.

Definition 4.2 (Resource types)**.** *A resource type is introduced by the* `resource` *keyword:*

$$R ::= \texttt{resource}\, T$$

Resource types are linear*: each value of a resource type must be consumed exactly once in every execution path.*

Definition 4.3 (Ownership qualifiers)**.** *JAPL provides two ownership qualifiers:*

(a) own T*: Exclusive ownership. The holder can read, write, and release the resource. Corresponds to the linear type T.*

(b) ref T*: Borrowed reference. The holder can read the resource but cannot write, release, or transfer it. Corresponds to $!T$ restricted to a scope.*

```
resource File
resource Socket

-- Opening a file acquires ownership (manual management via let)
fn read_config(path: Path) →Result[Config, ReadError] with Io =
  let file = File.open(path, Read)?
  let contents = File.read_all(file)?
  File.close(file) -- ownership consumed; compile error if omitted
  parse_config(contents)

-- Ownership transfer: send consumes the socket, returns it
fn send(socket: Owned[Socket], bytes: Bytes)
    → (Owned[Socket], Int) with Io =
  let n = Socket.write(socket, bytes)
  (socket, n)
```

Listing 2: Resource-layer code: explicit ownership

4.4 The Split in Practice

The boundary between pure and resource layers is clean and syntactically visible. Consider a web server:

```
-- Pure layer: all business logic
fn route(req: Request) →Response =
  match req.method, req.path with
  | Get, "/users/" ++id →get_user(id)
  | Post, "/users"     → create_user(req.body)
  | _                  → not_found()

fn get_user(id: String) →Response =
  match UserRepo.find(id) with
  | Some(user) →Response.json(200, User.to_json(user))
  | None      → Response.json(404, error_json("not found"))

-- Resource layer: IO boundary
fn serve(listener: Owned[TcpListener])
    → Never with Io, Process[ServerMsg] =
  let (listener, conn) = Tcp.accept(listener)
  let _ = Process.spawn(fn →handle_conn(conn))
  serve(listener)

fn handle_conn(conn: Owned[TcpSocket]) →Unit with Io =
  let (conn, req_bytes) = Tcp.read(conn)
  let req = parse_request(req_bytes) -- pure
  let resp = route(req)            -- pure
  let conn = Tcp.write(conn, serialize_response(resp))
  Tcp.close(conn)
```

Listing 3: The dual-layer split in a web server

In this example, the pure functions `route`, `get_user`, `parse_request`, and `serialize_response` make up the majority of the code and require no ownership annotations. Only `serve` and `handle_conn`, which interact with TCP sockets, operate in the resource layer.

4.5 Formal Typing Judgment

The dual-layer system uses a combined typing judgment:

Definition 4.4 (Dual-layer typing judgment)**.** *The judgment*

$$\Gamma; \Delta \vdash_{\varepsilon} e : \tau$$

asserts that expression e *has type* τ *under unrestricted (pure) context* Γ*, linear (resource) context* Δ*, and with effect* $\varepsilon \in \{\mathsf{Pure}, \mathsf{Io}, \ldots\}$*.*

The rules ensure:

(i) Variables in Γ *may be used any number of times.*

(ii) Variables in Δ *must each be used exactly once.*

(iii) Resource-producing operations (`File.open`, etc.) add bindings to Δ*.*

(iv) Resource-consuming operations (`File.close`, etc.) remove bindings from Δ*.*

(v) At every join point (if-then-else, match), the remaining Δ *must be identical in all branches.*

5 Ownership Semantics

5.1 Owned⟨T⟩

The type $\mathsf{Owned}\langle T\rangle$ represents exclusive ownership of a resource of type T. The owner has three rights:

1. **Use**: Perform operations on the resource (read, write).
2. **Transfer**: Pass ownership to another function or process.
3. **Release**: Deallocate the resource, releasing its underlying system resource.

These rights are mutually exclusive over time: once ownership is transferred or the resource is released, the original holder loses all rights.

Definition 5.1 (Ownership transfer)**.** *A function that accepts an* $\mathsf{Owned}\langle T\rangle$ *parameter* consumes *the caller's ownership. If the function returns an* $\mathsf{Owned}\langle T\rangle$*, ownership is transferred back to the caller. The typing rule is:*

$$\frac{\Gamma; \Delta, x : \mathsf{Owned}\langle T\rangle \vdash_{\varepsilon} e : \tau}{\Gamma; \Delta \vdash_{\varepsilon} \lambda(x : \mathsf{Owned}\langle T\rangle).\, e : \mathsf{Owned}\langle T\rangle \multimap_{\varepsilon} \tau} \textsc{ Own-Transfer}$$

5.2 Consumption and Return Patterns

Japl supports several patterns for resource consumption:

Terminal consumption. The resource is released and not returned:

```
fn close_file(file: Owned[File]) →Unit with Io =
  File.close(file)
  -- file is consumed; cannot be used after this point
```

Listing 4: Terminal consumption: File.close

Threaded consumption. The resource is used and then returned with updated state:

```
fn read_line(file: Owned[File])
    → (Owned[File], String) with Io =
  let (file, line) = File.read_line(file)
  (file, line)
```

Listing 5: Threaded consumption: read then return

Fork-join consumption. Ownership is temporarily split (borrowing) and then rejoined:

```
fn copy_file(src: Owned[File], dst: Owned[File])
    → (Owned[File], Owned[File]) with Io =
  let bytes = File.read_all(src)
  let dst = File.write_all(dst, bytes)
  (src, dst)
```

Listing 6: Fork-join via borrowing

5.3 Borrowing

Borrowing allows temporary read-only access to a resource without transferring ownership.

Definition 5.2 (Borrow). *A borrow of type* $\mathsf{ref}\ T$ *grants read-only access to a resource of type* T*. The borrow is valid only within the lexical scope of the borrowing expression. The typing rule is:*

$$\frac{\Gamma; \Delta, x : \mathsf{Owned}\langle T\rangle \vdash_{\varepsilon} e : \tau \qquad x \notin \mathrm{FV}(e) \textit{ as owned}}{\Gamma; \Delta, x : \mathsf{Owned}\langle T\rangle \vdash_{\varepsilon} \mathbf{borrow}\ x\ \mathbf{as}\ y\ \mathbf{in}\ [y/x]e : \tau}\ \textsc{Borrow}$$

where $y : \mathsf{ref}\ T$ *is the borrowed reference, and* x *remains available as* $\mathsf{Owned}\langle T\rangle$ *after the borrow scope exits.*

Remark 5.3 (Borrow desugaring). *The surface syntax* $\mathbf{borrow}\ x\ \mathbf{as}\ y\ \mathbf{in}\ e$ *desugars to the core calculus form* $\mathsf{borrow}(\mathsf{res}(a), \lambda y.\, e)$*, where* x *is bound to* $\mathsf{res}(a)$ *and the body* e *is wrapped in a lambda that receives the borrowed value* y*. The core reduction rule (Definition 10.4, rule* Borrow*) then reduces* $\mathsf{borrow}(\mathsf{res}(a), f)$ *to* $(f\ v,\ \mathsf{res}(a))$*, applying the closure to the live value* v *while preserving the resource handle.*

```
fn peek(buf: ref Buffer) →Byte =
  Buffer.get(buf, 0)

fn process(buf: Owned[Buffer]) →(Owned[Buffer], Byte) with Io =
  let byte = peek(ref buf) -- borrow for read-only access
  (buf, byte)              -- buf is still owned here
```

Listing 7: Borrowing: peek at a buffer without consuming it

The key constraint is that borrows cannot *outlive* the owner: the borrowed reference `ref` T is valid only within the scope where it is created. This is enforced statically by the compiler.

5.4 Region-Based Inference

Japl uses region-based inference [15] to automatically determine resource lifetimes and insert release operations.

Definition 5.4 (Region)**.** *A* region r *is an abstract lifetime associated with a lexical scope. Every resource allocation is assigned a region, and the resource is released when the region ends. Regions form a partial order: if scope r_1 is nested inside scope r_2, then $r_1 \sqsubseteq r_2$ (i.e., r_1 ends before r_2).*

The compiler's region inference pass operates in three phases:

1. **Constraint generation**: For each resource allocation, generate a region variable. For each use, generate a constraint that the resource's region must contain the use site. For each ownership transfer, propagate the region.
2. **Constraint solving**: Solve the region constraints to find the minimal region for each resource.
3. **Insertion**: Insert release operations at region boundaries where resources go out of scope.

Example 5.5 (Automatic release)**.** *In the following code, the compiler infers that* `conn` *must live at least as long as the scope of* `process_query`*:*

```
fn process_query(url: String, sql: String)
    → Result[Rows, DbError] with Io =
  use conn = Db.connect(url)?
  let result = Db.query(ref conn, sql)?
  -- compiler inserts: Db.close(conn) here
  Ok(result)
```

Listing 8: Region inference inserts Db.close automatically

The `use` *binding signals that* `conn` *is a linear resource; the compiler ensures it is released at the end of its region.*

5.5 Uniqueness Inference

Beyond region inference, Japl performs *uniqueness inference* to determine when a value has a single reference, enabling in-place mutation optimizations.

Definition 5.6 (Uniqueness)**.** *A value is* unique *if it has exactly one reference in the program state. A unique immutable value can be safely updated in-place (destructive update), because no other reference can observe the change.*

Proposition 5.7 (Uniqueness optimization)**.** *If a pure value v is unique at a record-update site $\{v \mid \ell = e\}$, the compiler may perform the update in-place, avoiding allocation. This is semantically equivalent to creating a fresh copy with the field changed.*

Proof. Since v has a single reference, no other expression can observe the difference between in-place update and copy-on-write. By parametricity, the program cannot distinguish the two implementations. □

This optimization is particularly important for the common pattern of threading state through recursive functions:

```
fn server_loop(state: ServerState) →Never with Process[Msg] =
  let msg = Process.receive()
  -- If state is unique (no other references), the update
  -- { state | count = state.count + 1 } is performed in-place.
  let new_state = { state | count = state.count + 1 }
  server_loop(new_state)
```

Listing 9: Uniqueness inference enables in-place update

6 Resource Types

6.1 Declaring Resource Types

Resource types are declared with the `resource` keyword:

```
resource File
resource Socket
resource DbConnection
resource GpuBuffer
resource MutableArray[a]
```

Listing 10: Resource type declarations

The `resource` keyword instructs the compiler to apply linear typing rules to values of this type. A resource cannot be silently dropped, duplicated, or aliased.

6.2 Resource Lifecycle

Every resource follows a lifecycle:

1. **Acquisition**: A resource is created via a constructor (e.g., `File.open`, `Tcp.connect`). The constructor returns $\mathsf{Owned}\langle R\rangle$.

2. **Use**: The resource is operated on via functions that take $\mathsf{Owned}\langle R\rangle$ or $\mathsf{ref}\ R$. Functions that take $\mathsf{Owned}\langle R\rangle$ must return it (threaded pattern) or consume it (terminal pattern).

3. **Release**: The resource is released via a destructor (e.g., `File.close`, `Tcp.disconnect`). The destructor consumes $\mathsf{Owned}\langle R\rangle$ and returns no resource.

Definition 6.1 (Resource safety)**.** *A program is* resource-safe *if, for every resource acquired during execution, the resource is eventually released exactly once.*

Theorem 6.2 (Well-typed programs are resource-safe)**.** *If $\Gamma; \cdot \vdash_{\varepsilon} e : \tau$ (i.e., the program type-checks with an empty linear context), then execution of e is resource-safe.*

We defer the proof to Section 10.

6.3 The `use` Binding

The **`use`** keyword introduces a linear binding with automatic release semantics:

```
fn read_entire_file(path: String) →Result[String, IoError] with Io =
  use file = File.open(path, Read)?
  let contents = File.read_all(file)?
  -- File.close(file) is automatically inserted by the compiler
  -- at the end of the ‘use‘ scope
  Ok(contents)
```

Listing 11: The **`use`** binding

The **`use`** binding desugars to a try-finally pattern: even if the body raises an error (via the `?` operator or a crash), the resource is released.

Definition 6.3 (**`use`** desugaring)**.** *The binding* **`use`** $x = e_1$ ***in*** e_2 *desugars to:*

$$\textbf{let } x = e_1 \textbf{ in try } e_2 \textbf{ finally } \mathit{release}(x)$$

where release is the destructor associated with the resource type of x*.*

6.4 Parametric Resource Types

Resources can be parametric:

```
resource MutableArray[a]

fn sort_in_place(arr: own MutableArray[Int])
    → own MutableArray[Int] =
  let len = MutableArray.length(arr)
  loop i = 0, arr = arr while i < len do
    loop j = i + 1, arr = arr while j < len do
      if MutableArray.get(arr, j) < MutableArray.get(arr, i) then
        let arr = MutableArray.swap(arr, i, j)
        continue(j + 1, arr)
      else
        continue(j + 1, arr)
    continue(i + 1, arr)
  arr
```

Listing 12: Parametric resource types

The linearity constraint applies to the outer resource container, not to its type parameter. The elements of a `MutableArray[Int]` are pure `Int` values and can be freely copied.

6.5 Freezing: Resource to Value Conversion

A resource can be *frozen* to produce an immutable value, transferring data from the resource layer to the pure layer:

```
fn build_message() →Bytes with Io =
  let buf = Buffer.alloc(1024)       -- resource layer
  let buf = Buffer.write(buf, 0, header)
  let buf = Buffer.write(buf, 64, payload)
  Buffer.freeze(buf)                  -- pure layer: immutable Bytes
  -- buf is consumed; only the frozen Bytes value remains
```

Listing 13: Freezing a buffer into immutable bytes

Definition 6.4 (Freeze)**.** *A* freeze *operation has type* Owned$\langle R\rangle \multimap \tau$*, consuming the resource and producing a pure value. After freezing, the resource is released and the returned value lives in the pure layer (managed by GC).*

Freezing corresponds to the functor $G : \mathcal{L} \to \mathcal{C}$ in the LNL model (Theorem 3.6): it maps a linear resource to an unrestricted pure value.

7 Interaction Between Layers

7.1 Pure Functions Using Resources

Pure functions cannot *own* resources, but they can *borrow* them. A function that borrows a resource via ref T does not participate in the ownership protocol; it simply reads the resource's current state.

```
-- This function is pure except for the borrow
fn buffer_checksum(buf: ref Buffer) →Int =
  let bytes = Buffer.to_bytes_view(buf) -- read-only view
  Bytes.fold(bytes, 0, fn acc, b →acc + b)

fn validate_and_send(buf: Owned[Buffer], sock: Owned[Socket])
    → (Owned[Buffer], Owned[Socket]) with Io =
  let sum = buffer_checksum(ref buf) -- borrow for pure computation
  if sum > 0 then
    let (sock, _) = Socket.send(sock, Buffer.to_bytes(ref buf))
    (buf, sock)
  else
    (buf, sock)
```

Listing 14: Pure function borrowing a resource

7.2 Ownership Transfer Between Processes

When a resource is sent to another process, ownership transfers:

```
fn producer() →Unit with Io, Process[ProducerMsg] =
  let buf = Buffer.alloc(4096)
  let buf = Buffer.write(buf, 0, generate_data())
  -- Ownership transfers to the consumer process
  Process.send(consumer_pid, ProcessBuffer(buf))
  -- buf is consumed; using it here is a compile error

fn consumer() →Never with Io, Process[ConsumerMsg] =
  match Process.receive() with
  | ProcessBuffer(buf) →
      -- We now own buf
      let data = Buffer.freeze(buf)
```

```
    process_data(data)
    consumer()
```

Listing 15: Ownership transfer between processes

7.3 The Escape Hatch

For interoperability with C libraries and low-level systems code, JAPL provides an `unsafe` escape hatch that allows bypassing linearity checks:

```
-- unsafe block: linearity is not checked
fn raw_mmap(fd: Owned[File], size: Int) →Owned[MappedRegion] with Io =
  unsafe {
    let ptr = foreign_mmap(File.raw_fd(ref fd), size)
    MappedRegion.from_raw(ptr, size)
  }
  -- Outside unsafe: normal linearity rules apply
```

Listing 16: The unsafe escape hatch

The `unsafe` keyword has several properties:

1. It must appear explicitly in the source code, making it searchable and auditable.
2. It does not propagate: a function containing `unsafe` can still present a safe interface to its callers.
3. It is tracked by the effect system: `unsafe` blocks require the `Unsafe` effect, which cannot be silently introduced.

7.4 Effect System Interaction

Resource operations are effectful: they carry the `Io` effect. The effect system and the ownership system cooperate:

Definition 7.1 (Effect-ownership interaction)**.** *The following invariants hold:*

(i) Resource acquisition requires an effect (typically Io*).*

(ii) Resource release requires the same effect as acquisition.

(iii) Pure functions ($\varepsilon = \mathsf{Pure}$) cannot acquire or release resources; they can only borrow.

(iv) Effect handlers cannot duplicate or discard resources: the handler's continuation must preserve the linear context.

Proposition 7.2 (Effect handlers preserve linearity)**.** *If an effect handler h transforms computation $\Gamma; \Delta \vdash_\varepsilon e : \tau$ into $\Gamma; \Delta' \vdash_{\varepsilon'} e' : \tau'$, then $\Delta' = \Delta$ (the linear context is unchanged).*

Proof. By induction on the structure of the effect handler. The handler provides implementations for effect operations, but each operation implementation must preserve the linear context: it receives a continuation that expects the same linear resources, and it must pass exactly those resources to the continuation. Since the handler cannot forge or destroy linear resources (this is enforced by the !-I rule, which prevents linear resources from appearing in unrestricted positions), the linear context is preserved. □

8 Comparison with Related Languages

8.1 Overview

Table 1: Comparison of mutation control strategies across languages

Language	Mutation model	Resource mgmt	Annotation	GC
Japl	Dual-layer	Linear ownership	Low (pure layer)	Hybrid
Rust	Ownership+borrow	Ownership+RAII	High (everywhere)	None
Go	Unrestricted	GC + `defer`	None	Full GC
Haskell	IO monad	GC + finalizers	Medium (monadic)	Full GC
Erlang	Copy everything	Per-process GC	None	Per-proc GC
OCaml	Mutable refs	GC	Low	Full GC
Clean	Uniqueness types	Uniqueness	Medium	GC

8.2 Rust: Full Borrow Checker

Rust applies its ownership and borrowing discipline to *every* value, including purely functional data. This provides strong guarantees—no GC is needed, and memory safety is ensured at compile time—but imposes a significant annotation burden.

Consider a simple data transformation in Rust:

```
fn transform(data: Vec<i32>) -> Vec<i32> {
    data.into_iter().map(|x| x * 2).collect()
}
```

The programmer must choose between `Vec<i32>` (ownership transfer), `&Vec<i32>` (immutable borrow), and `&mut Vec<i32>` (mutable borrow). For a pure transformation, this choice is an unnecessary cognitive burden.

In Japl, the same function requires no ownership annotations:

```
fn transform(data: List[Int]) →List[Int] =
  List.map(data, fn x →x * 2)
```

Listing 17: Japl vs. Rust: pure data transformation

The trade-off is that Japl's pure layer requires a garbage collector, while Rust avoids GC entirely. We argue that this trade-off is worthwhile for application-level code, where GC pauses are negligible relative to I/O latency.

8.3 Go: GC Everything

Go takes the opposite extreme: all memory is garbage-collected, and there is no ownership system. Resources are managed by convention (`defer` and the `io.Closer` interface) rather than by the type system.

The consequence is that resource leaks are possible:

```
func processFile(path string) error {
    f, err := os.Open(path)
    if err != nil { return err }
    // Easy to forget: defer f.Close()
    data, err := io.ReadAll(f)
    // If we return here, f is leaked
    ...
}
```

Japl's linear types make this class of bug impossible: the compiler rejects any program that fails to consume a resource.

8.4 Haskell: The IO Monad

Haskell confines side effects to the `IO` monad, ensuring that pure functions are referentially transparent. However, the monadic style has several drawbacks:

1. Monadic binding (`>>=`) is syntactically heavier than direct-style code.
2. Composing different effects (state, error, IO) requires monad transformers, which are notoriously difficult to use.
3. Resources within `IO` are not linearly tracked; finalizers and `bracket` provide safety but not compile-time guarantees.

Japl's effect system provides effect tracking without monadic syntax: effects are listed after `with` in function signatures, and composition is automatic (effects form a commutative monoid). The resource layer adds linear tracking that Haskell's `IO` lacks.

8.5 Erlang: Copy Everything

Erlang achieves concurrency safety by making all values immutable and copying data between processes. This eliminates shared mutable state entirely but has performance implications for large data transfers.

Japl inherits Erlang's process isolation model for pure values but extends it with ownership transfer for resources: a resource can be moved between processes without copying, while maintaining the invariant that only one process owns it at a time.

8.6 OCaml: Mutable Refs

OCaml allows mutation through mutable record fields and `ref` cells. The type system does not distinguish pure and impure code; any function may have side effects.

This pragmatic approach simplifies the type system but sacrifices the ability to reason about purity. Japl's effect annotations (`with Io`, `with State[s]`) restore this reasoning while keeping the annotation burden low (effects are inferred within function bodies).

8.7 Clean: Uniqueness Types

Clean's uniqueness types [13] are the closest relative of Japl's resource layer. Clean uses a `*` attribute to mark unique values; the compiler ensures that unique values are not aliased.

Japl extends Clean's approach in several ways:

1. **Explicit layer separation**: Japl does not apply uniqueness to all types; only declared `resource` types are subject to linear discipline.

2. **Borrowing**: Clean does not have a borrowing mechanism; in Clean, any use of a unique value consumes it. Japl's `ref` qualifier allows non-consuming reads.

3. **Effect integration**: Japl's resource operations are annotated with effects, providing an additional layer of safety.

4. **Process integration**: Japl supports ownership transfer between processes, which Clean (not being an actor-based language) does not address.

8.8 Summary of Trade-offs

Table 2: Trade-off analysis: safety guarantees vs. programmer burden

	Data race free	Resource safe	No GC needed	Low annotation	Effect tracking
Japl	✓	✓		✓	✓
Rust	✓	✓	✓		
Go				✓	
Haskell	✓				✓
Erlang	✓			✓	
OCaml				✓	
Clean	✓	✓			

Japl is the only language in this comparison that simultaneously provides data-race freedom, compile-time resource safety, low annotation burden (for the majority of code), and effect tracking. The cost is a garbage collector for the pure layer; we argue in Section 11 that this is an acceptable trade-off.

9 Implementation

9.1 Architecture Overview

The Japl compiler and runtime implement the dual-layer model as follows:

1. **Parser**: Produces an AST with explicit `use`, `own`, and `ref` annotations.

2. **Type checker**: Performs bidirectional type checking with local inference. Maintains separate unrestricted (Γ) and linear (Δ) contexts.

3. **Linearity checker**: Verifies that all linear variables are consumed exactly once on every execution path.

4. **Region inference**: Assigns regions to resource allocations and inserts release operations.

5. **Uniqueness analysis**: Identifies pure values with a single reference for in-place update optimization.

6. **Code generation**: Compiles to native code via LLVM or Cranelift.
7. **Runtime**: Manages the dual-layer memory model at execution time.

9.2 Type Checker Implementation

The type checker uses a *splitting* judgment to partition the linear context among subexpressions [5]:

Definition 9.1 (Context splitting)**.** *A* split *of linear context Δ into Δ_1 and Δ_2 (written $\Delta = \Delta_1 \uplus \Delta_2$) assigns each variable in Δ to exactly one of Δ_1 or Δ_2.*

For function application $e_1\ e_2$, the type checker:

1. Generates a fresh split of the current linear context.
2. Checks e_1 against Δ_1.
3. Checks e_2 against Δ_2.
4. Verifies that Δ_1 and Δ_2 are both fully consumed.

For conditionals and pattern matching, the type checker verifies that all branches consume the *same* set of linear variables:

$$\frac{\Gamma; \Delta_0 \vdash e : \tau \qquad \forall i.\ \Gamma; \Delta_i \vdash p_i \Rightarrow e_i : \sigma \qquad \Delta_1 = \Delta_2 = \cdots = \Delta_n}{\Gamma; \Delta_0, \Delta_1 \vdash \textbf{match}\ e\ \textbf{with}\ \{p_i \Rightarrow e_i\} : \sigma}\ \textsc{Match}$$

9.3 Linearity Checking Algorithm

The linearity checker runs as a separate pass after type checking. It traverses the typed AST and maintains a set of "live" linear variables. The algorithm is:

1. At a `let` binding of a resource type: add the variable to the live set.
2. At a use of a linear variable: remove it from the live set. If already removed, report a "use after move" error.
3. At a branch point (if/match): check each branch independently; verify that all branches have the same live set at exit.
4. At function exit: verify that the live set matches the function's return type (i.e., any linear variable still live must be returned).
5. At scope exit for `use` bindings: if the variable is still live, insert a release call.

9.4 Runtime Memory Model

The runtime maintains two memory regions per process:

Immutable heap (GC-managed). All pure values reside here. The garbage collector is generational and per-process:

- **Nursery**: Small, per-process bump allocator. Most values die young and are collected cheaply.
- **Old generation**: Surviving values are promoted to a shared old generation, collected concurrently.
- **No write barriers**: Since data is immutable, there are no pointer mutations to track. This significantly simplifies the GC.
- **Process death**: When a process terminates, its entire nursery is reclaimed instantly.

Resource arena (ownership-managed). All linear resources reside here. There is no garbage collection; resources are freed deterministically when consumed:

- **Allocation**: Resources are allocated in a per-process arena.
- **Deallocation**: When a resource is consumed (released), its memory is immediately freed.
- **Transfer**: When ownership is transferred to another process, the resource is moved from one process's arena to another's.
- **No fragmentation**: Arena allocation avoids the fragmentation problems of general-purpose allocators.

9.5 Compile-Time Ownership Verification

The compiler performs ownership verification as a fixed-point computation over the control-flow graph (CFG):

1. Build the CFG for each function.
2. At each CFG node, compute the set of live linear variables.
3. At each join point, check that the live sets from all predecessors are identical.
4. At the function exit node, check that the live set matches the expected output (linear variables that appear in the return type).
5. If any check fails, emit a diagnostic with the offending variable and the point where it was consumed or escaped.

This analysis is similar to liveness analysis in traditional compilers but with the additional constraint that each linear variable must be consumed *exactly once*, not "at least once" or "at most once."

9.6 Performance Characteristics

The dual-layer design has favorable performance characteristics:

- **Pure-layer overhead**: GC pauses are bounded by the nursery size and are per-process (no global stop-the-world). Uniqueness inference eliminates unnecessary copies.
- **Resource-layer overhead**: Zero runtime overhead beyond the actual resource operations; ownership is verified entirely at compile time.
- **Combined**: The compiler can optimize across layers. For example, a buffer that is allocated, filled, and frozen within a single function can be stack-allocated if uniqueness analysis proves it does not escape.

10 Formal Proofs

This section proves type safety and resource safety for JAPL's dual-layer system. We work with a core calculus λ^{JR} (JAPL Resources) that captures the essential features.

10.1 Syntax of λ^{JR}

Definition 10.1 (Core calculus λ^{JR})**.**

$$\begin{aligned}
\textit{Types}\quad & \tau ::= B \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid R \mid \mathsf{Owned}\langle R\rangle \\
\textit{Expressions}\quad & e ::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_i(e) \\
& \quad\ \mid \mathsf{new}_R \mid \mathsf{use}(e_1, e_2) \mid \mathsf{release}(e) \mid \mathsf{borrow}(e_1, e_2) \\
& \quad\ \mid \mathsf{freeze}(e) \mid v \\
\textit{Values}\quad & v ::= \lambda x.\, e \mid (v_1, v_2) \mid c \mid r \\
\textit{Resources}\quad & r ::= \mathsf{res}(a) \quad \textit{(resource at address a)}
\end{aligned}$$

where B ranges over base types and R ranges over resource type names.

Remark 10.2 (Effects in λ^{JR})**.** *The core calculus λ^{JR} deliberately omits effect tracking (*Io*,* $\mathsf{State}[s]$*, etc.) to isolate the contribution of linear ownership. Effect annotations appear in the surface-language typing rules (Theorem 4.4) and in the resource-specific rules (T-New, T-Release, T-Freeze), but a full formalization of the effect system—including effect rows, handler semantics, and the interaction between effect polymorphism and linearity—is the subject of a companion paper on JAPL's algebraic effect system. The two systems compose orthogonally: linearity governs* how many times *a resource is used, while effects govern* what operations *may be performed. Theorem 7.2 states the key compatibility property between the two.*

Definition 10.3 (Store)**.** *A* store σ *is a partial function from addresses to resource states:*

$$\sigma : \mathsf{Addr} \rightharpoonup \mathsf{ResourceState}$$

where $\mathsf{ResourceState} ::= \mathsf{Live}(v) \mid \mathsf{Released}$.

10.2 Operational Semantics

We write $\bot_R$ for the *default initial state* of resource type R: the distinguished value that a freshly allocated resource holds before any operations have been performed on it. Each resource type declaration implicitly defines its $\bot_R$ (e.g., an empty buffer, an unconnected socket descriptor).

Definition 10.4 (Small-step reduction)**.** *The reduction relation $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ is defined by:*

$$\frac{}{\langle (\lambda x.\, e)\ v, \sigma \rangle \longrightarrow \langle [v/x]e, \sigma \rangle}\ \textit{App} \qquad \frac{a \notin \mathrm{dom}(\sigma)}{\langle \mathsf{new}_R, \sigma \rangle \longrightarrow \langle \mathsf{res}(a), \sigma[a \mapsto \mathsf{Live}(\bot_R)] \rangle}\ \textit{New}$$

$$\frac{\sigma(a) = \mathsf{Live}(v)}{\langle \mathsf{release}(\mathsf{res}(a)), \sigma \rangle \longrightarrow \langle (), \sigma[a \mapsto \mathsf{Released}] \rangle}\ \textit{Release}$$

$$\frac{\sigma(a) = \mathsf{Live}(v)}{\langle \mathsf{freeze}(\mathsf{res}(a)), \sigma \rangle \longrightarrow \langle v, \sigma[a \mapsto \mathsf{Released}] \rangle}\ \textit{Freeze}$$

$$\frac{\sigma(a) = \mathsf{Live}(v)}{\langle \mathsf{borrow}(\mathsf{res}(a), f), \sigma \rangle \longrightarrow \langle (f\ v, \mathsf{res}(a)), \sigma \rangle}\ \textit{Borrow}$$

10.3 Type Safety

Definition 10.5 (Store typing)**.** *A store typing Σ is a partial function from addresses to resource types:*

$$\Sigma : \mathsf{Addr} \rightharpoonup \{\mathsf{Live}(R) \mid \mathsf{Released}\}$$

We write $\sigma : \Sigma$ when for every $a \in \mathrm{dom}(\Sigma)$:

- *If $\Sigma(a) = \mathsf{Live}(R)$, then $\sigma(a) = \mathsf{Live}(v)$ for some v of type R.*
- *If $\Sigma(a) = \mathsf{Released}$, then $\sigma(a) = \mathsf{Released}$.*

Theorem 10.6 (Progress)**.** *If $\Gamma; \Delta \vdash_\varepsilon e : \tau$ and $\sigma : \Sigma$, then either:*

(a) e is a value, or

(b) there exist e', σ' such that $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$.

Proof. By induction on the typing derivation $\Gamma; \Delta \vdash_\varepsilon e : \tau$.

Case Var-Lin: $e = x$ and $x : \mathsf{Owned}\langle R \rangle \in \Delta$. Then x is bound to a value (a resource reference $\mathsf{res}(a)$), so e is a value.

Case Var-Un: Similar.

Case $\multimap$-I: $e = \lambda x.\, e'$, which is a value.

Case $\multimap$-E: $e = e_1\ e_2$. By induction, either e_1 is a value or it can step. If e_1 can step, so can e. If e_1 is a value, it has type $A \multimap B$, so it is a lambda $\lambda x.\, e'$. By induction, either e_2 is a value or it can step. If e_2 can step, so can e. If e_2 is a value v, then e can step by the App rule.

Case New: $e = \mathsf{new}_R$. We can always find a fresh address $a \notin \mathrm{dom}(\sigma)$ (addresses are unbounded), so the New rule applies.

Case Release: $e = \mathsf{release}(\mathsf{res}(a))$. By the typing rule, $\mathsf{res}(a) : \mathsf{Owned}\langle R \rangle$, so $a \in \mathrm{dom}(\Sigma)$ with $\Sigma(a) = \mathsf{Live}(R)$. Since $\sigma : \Sigma$, we have $\sigma(a) = \mathsf{Live}(v)$, so the Release rule applies.

The remaining cases (Freeze, Borrow, pairs, projections) follow similarly. ☐

Theorem 10.7 (Preservation)**.** *If $\Gamma; \Delta \vdash_\varepsilon e : \tau$ and $\sigma : \Sigma$ and $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$, then there exist Γ', Δ', Σ' such that:*

(a) $\Gamma'; \Delta' \vdash_\varepsilon e' : \tau$

(b) $\sigma' : \Sigma'$

(c) $\Sigma \subseteq \Sigma'$ *(the store typing only grows or transitions* Live *entries to* Released*)*

Proof. By induction on the typing derivation, with case analysis on the reduction rule applied.

Case App: $e = (\lambda x.\, e')\ v$ steps to $[v/x]e'$. By the T-App rule, the linear context is split as $\Delta = \Delta_1 \uplus \Delta_2$ with $\Gamma; \Delta_1 \vdash (\lambda x.\, e') : \tau_1 \multimap \tau$ and $\Gamma; \Delta_2 \vdash v : \tau_1$. By the substitution lemma for the dual-layer system (Theorem 10.8), $\Gamma; \Delta' \vdash [v/x]e' : \tau$ where $\Delta' = \Delta_1 \uplus (\Delta_2 \setminus \{x : \tau_1\})$, i.e., Δ with x's binding consumed. The store is unchanged.

Case New: $e = \mathsf{new}_R$ steps to $\mathsf{res}(a)$ with $\sigma' = \sigma[a \mapsto \mathsf{Live}(\bot_R)]$. Set $\Sigma' = \Sigma[a \mapsto \mathsf{Live}(R)]$. Then $\mathsf{res}(a) : \mathsf{Owned}\langle R \rangle$ under $\Delta' = \Delta, a : \mathsf{Owned}\langle R \rangle$, and $\sigma' : \Sigma'$.

Case Release: $e = \mathsf{release}(\mathsf{res}(a))$ steps to $()$ with $\sigma' = \sigma[a \mapsto \mathsf{Released}]$. Set $\Sigma' = \Sigma[a \mapsto \mathsf{Released}]$. The result $()$ has type `Unit`. The linear context $\Delta' = \Delta \setminus \{a : \mathsf{Owned}\langle R \rangle\}$: the binding is consumed.

Case Freeze: Similar to Release, but the result is the value v stored in the resource, which has a pure type.

Case Borrow: $e = \mathsf{borrow}(\mathsf{res}(a), f)$ steps to $(f\ v, \mathsf{res}(a))$. The resource remains live; the linear binding is preserved. The result is a pair of the function application result and the resource.

The remaining cases are standard. □

Lemma 10.8 (Substitution)**.** *If $\Gamma; \Delta, x : A \vdash_\varepsilon e : B$ and $\Gamma; \cdot \vdash v : A$ (if A is unrestricted) or $v = \mathsf{res}(a)$ with $a : \mathsf{Owned}\langle R \rangle$ (if A is linear), then $\Gamma; \Delta' \vdash_\varepsilon [v/x]e : B$ where:*

- $\Delta' = \Delta$ *if A is unrestricted.*
- $\Delta' = \Delta$ *if A is linear and x is consumed exactly once in e.*

Proof. By induction on the typing derivation, tracking the number of occurrences of x in e. The linearity checker ensures exactly one occurrence for linear variables. □

10.4 Resource Safety

Theorem 10.9 (Resource safety)**.** *If $\Gamma; \cdot \vdash_\varepsilon e : \tau$ (the program type-checks with an empty linear context) and $\langle e, \emptyset \rangle \longrightarrow^* \langle v, \sigma \rangle$ (the program terminates with value v and final store σ), then for every $a \in \mathrm{dom}(\sigma)$:*

$$\sigma(a) = \mathsf{Released}$$

That is, every resource acquired during execution has been released.

Proof. We establish the invariant: at every step of reduction, the number of Live entries in σ equals the size of the linear context Δ.

Base case: Initially, $\sigma = \emptyset$ and $\Delta = \cdot$, so the invariant holds trivially.

Inductive step: By preservation (Theorem 10.7), each reduction step preserves the correspondence between Δ and Σ. Specifically:

- New: adds one entry to both σ (Live) and Δ, preserving the count.

- Release/Freeze: transitions one entry in σ from Live to $\mathsf{Released}$ and removes one entry from Δ, preserving the count.
- All other rules: preserve both σ and the size of Δ.

Conclusion: When the program terminates with $\Gamma; \cdot \vdash v : \tau$ (empty linear context), the number of Live entries in σ is zero. Therefore every entry is $\mathsf{Released}$. □

Corollary 10.10 (No resource leaks)**.** *Well-typed* JAPL *programs (without* `unsafe`*) cannot leak resources: every file is closed, every socket is disconnected, every buffer is freed.*

Corollary 10.11 (No use-after-free)**.** *Well-typed* JAPL *programs cannot use a resource after it has been released: the linear type system ensures that the binding is consumed by the release operation and cannot be used again.*

Corollary 10.12 (No double-free)**.** *Well-typed* JAPL *programs cannot release a resource twice: the linear type system ensures exactly-once consumption.*

11 Discussion

11.1 When Ownership Pressure Leaks

The dual-layer design succeeds when the boundary between pure and resource code is clean. In practice, however, there are cases where ownership concerns propagate into otherwise-pure code:

Resource-carrying data structures. If a data structure contains a resource (e.g., a list of open file handles), the entire structure becomes linear. This forces the programmer to manage ownership for the container, even if the business logic only cares about the data inside.

Callbacks with resources. A higher-order function that accepts a callback may need to thread resources through the callback, complicating the interface.

Long-lived resources. A database connection pool manages resources with lifetimes that span many function calls. The pool itself becomes a resource, and every function that uses the pool must participate in the ownership protocol.

11.2 Mitigations

JAPL provides several mechanisms to reduce ownership pressure:

Borrowing (`ref`). Most pure functions that need to read a resource can take a borrowed reference, avoiding ownership transfer entirely.

Scoped resources (`use`). The `use` binding automatically manages resource lifetimes within a scope, reducing boilerplate.

Process encapsulation. Long-lived resources can be owned by a dedicated process, which exposes a pure message-passing interface. Other processes interact with the resource through messages, never seeing the ownership:

```
-- The database process owns the connection
fn db_process(conn: Owned[DbConnection])
    → Never with Io, Process[DbMsg] =
  match Process.receive() with
  | Query(sql, reply) →
      let (conn, result) = Db.query(conn, sql)
      Reply.send(reply, result)
      db_process(conn)
  | Shutdown →
      Db.close(conn)
      Process.exit(Normal)

-- Callers never see Owned[DbConnection]
fn get_user(db: Pid[DbMsg], id: UserId)
    → Result[User, DbError] with Process =
  let result = Process.call(db, fn reply →Query(user_sql(id), reply))
  parse_user_row(result)
```

Listing 18: Process encapsulation hides ownership

This pattern is the recommended way to manage long-lived resources in Japl: the process boundary naturally encapsulates the ownership, and the caller's interface is pure message-passing.

Region-based grouping. Multiple resources acquired in the same scope can be grouped into a single region, reducing the number of individual ownership threads.

11.3 The 80/20 of Resource Management

Our experience with prototype Japl applications suggests the following distribution:

- **80%** of code is pure: data transformations, business logic, protocol handling, serialization. No ownership annotations needed.
- **15%** of code uses scoped resources (`use`): open file, read, close; connect, query, disconnect. Ownership is automatic via the `use` binding.
- **5%** of code requires explicit ownership management: long-lived resources, cross-process transfers, complex resource graphs. This is where the linear type system provides its greatest value.

The dual-layer design ensures that the 80% majority pays no annotation cost, while the 5% minority gets strong compile-time guarantees.

11.4 Effect System Synergy

The effect system and the ownership system reinforce each other. The effect system ensures that resource operations are visible in function signatures (`with Io`), preventing hidden side effects. The ownership system ensures that resources are properly managed even when effects are composed.

Together, they provide a stronger guarantee than either alone: a function with signature `fn(x: Int) -> Int` is guaranteed to be pure—it cannot secretly open a file, allocate a buffer, or access the network.

11.5 Comparison with Monadic IO

The Haskell approach confines *all* effects to the `IO` monad, including both resource management and general side effects. This conflates two concerns:

1. **Effect tracking**: Knowing that a function performs IO.
2. **Resource management**: Ensuring that resources are properly released.

Japl separates these concerns: the effect system handles (1), and the ownership system handles (2). This separation allows finer-grained reasoning: two functions may both have effect `Io`, but one manages resources (visible in its type via `Owned`) while the other merely reads a file.

11.6 Limitations

The dual-layer approach has several limitations:

1. **GC dependency**: The pure layer requires a garbage collector, making Japl unsuitable for environments where GC is unacceptable (embedded systems, real-time audio).
2. **Layer crossing cost**: Moving data between layers (freezing a buffer into immutable bytes) may involve a copy if the resource is not uniquely held.
3. **Learning curve**: Programmers must understand the distinction between pure and resource types, even if the pure layer requires no annotations.
4. **Expressiveness**: Some patterns that are natural in Rust (e.g., self-referential structures with lifetimes) are more difficult to express in Japl's simpler ownership model.

11.7 Future Work

Several directions remain for future investigation:

1. **Graded linearity**: Allowing resources to be used exactly n times (useful for connection pools with bounded concurrency).
2. **Session types**: Extending resource types with session type protocols, ensuring that resources are used in the correct order (e.g., connect, authenticate, query, disconnect).
3. **Dependent resource types**: Allowing resource types to depend on values, enabling richer invariants (e.g., a file handle that tracks whether it is open for reading, writing, or both).
4. **Formal verification**: Connecting Japl's type system to separation logic via the LNL model, enabling machine-checked proofs of resource safety.

12 Conclusion

We have presented Japl's second core principle: **mutation is local and explicit**. The dual-layer type system—pure values managed by GC, linear resources managed by ownership—achieves a design point that has not been explored by prior languages:

1. **Safety without pervasive annotation**: The majority of code lives in the pure layer and requires no ownership annotations, lifetime parameters, or borrow-checker reasoning.
2. **Resource safety by construction**: Every resource acquired by a well-typed program is released exactly once, as proven by our type safety and resource safety theorems.
3. **Effect visibility**: The effect system makes mutation visible in function signatures, enabling equational reasoning for pure code and local reasoning for effectful code.
4. **Process integration**: Ownership transfer between processes extends the linear discipline to concurrent settings, while process encapsulation hides ownership behind pure message-passing interfaces.

The formal framework—rooted in Girard's linear logic, Benton's LNL adjunction model, and the theory of substructural type systems—provides a rigorous foundation for the design. The categorical interpretation (Section 3.3) shows that Japl's dual-layer system is not an *ad hoc* engineering compromise but a principled instantiation of the well-studied relationship between linear and Cartesian closed categories.

We believe that the dual-layer approach represents a practical sweet spot for language design: it gives systems programmers the control they need over resources, while giving application programmers the simplicity they want for everyday code. The principle that mutation should be local and explicit—not absent, not pervasive, but *controlled*—is central to Japl's mission of being "pure by default, concurrent by design, resource-safe by construction, distributed without apology."

References

[1] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[2] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, pages 561–581. North-Holland, 1990.

[3] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of LNCS, pages 185–210. Springer, 1993.

[4] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286, 2012.

[5] David Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–43. MIT Press, 2005.

[6] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[7] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages.* MIT Press, 2004.

[8] P. Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic (CSL)*, volume 933 of LNCS, pages 121–135. Springer, 1995.

[9] Michael Barr. $*$-Autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1(2):159–178, 1991.

[10] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2018.

[11] The Rust Team. *The Rust Programming Language.* No Starch Press, 2021. `https://doc.rust-lang.org/book/`.

[12] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[13] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[14] Rinus Plasmeijer and Marko van Eekelen. *Clean Language Report, version 2.1.* University of Nijmegen, 2001.

[15] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[16] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, 2002.

[17] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.

[18] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[19] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, 2006.

[20] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems (ESOP)*, volume 5502 of LNCS, pages 80–94. Springer, 2009.

[21] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[22] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 145–158, 2013.

[23] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[24] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2003.

[25] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[26] Xavier Leroy. *The OCaml System: Documentation and User's Manual*. INRIA, 2000.

[27] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015.

[28] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12, 2015.

[29] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 447–458, 2011.

[30] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 486–499, 2017.

[31] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F$^\circ$. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pages 77–88, 2010.

[32] Neel Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[33] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

[34] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[35] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983.

[36] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[37] Fernando Borretti. The Austral programming language. `https://austral-lang.org/`, 2023.

[38] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, 1993.

[39] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 476–489, 2016.

A Complete Typing Rules for λ^{JR}

For reference, we present the complete set of typing rules for the core calculus.

Definition A.1 (Full typing rules)**.**

$$\frac{x : \tau \in \Gamma}{\Gamma; \cdot \vdash x : \tau}\ \textit{T-Var-Un} \qquad \frac{}{\Gamma; x : \tau \vdash x : \tau}\ \textit{T-Var-Lin} \qquad \frac{\Gamma; \Delta, x : \tau_1 \vdash e : \tau_2}{\Gamma; \Delta \vdash \lambda x.\, e : \tau_1 \multimap \tau_2}\ \textit{T-Abs}$$

$$\frac{\Gamma, x : \tau_1; \cdot \vdash e : \tau_2}{\Gamma; \cdot \vdash \lambda x.\, e : \tau_1 \to \tau_2}\ \textit{T-Abs-Un} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_1}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\ e_2 : \tau_2}\ \textit{T-App}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_1}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\ e_2 : \tau_2}\ \textit{T-App-Un} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}\ \textit{T-Pair}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad \Gamma; \Delta_2, x : \tau_1, y : \tau_2 \vdash e_2 : \tau}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let}\ (x, y) = e_1\ \textbf{in}\ e_2 : \tau}\ \textit{T-LetPair}$$

$$\frac{}{\Gamma; \cdot \vdash_{\mathsf{Io}} \mathsf{new}_R : \mathsf{Owned}\langle R\rangle}\ \textit{T-New} \qquad \frac{\Gamma; \Delta \vdash e : \mathsf{Owned}\langle R\rangle}{\Gamma; \Delta \vdash_{\mathsf{Io}} \mathsf{release}(e) : \mathsf{Unit}}\ \textit{T-Release}$$

$$\frac{\Gamma; \Delta \vdash e : \mathsf{Owned}\langle R\rangle \qquad R \hookrightarrow \tau}{\Gamma; \Delta \vdash_{\mathsf{Io}} \mathsf{freeze}(e) : \tau}\ \textit{T-Freeze}$$

$$\frac{\Gamma; \Delta, x : \mathsf{Owned}\langle R\rangle \vdash e_1 : \mathsf{Owned}\langle R\rangle \qquad \Gamma; y : \mathsf{ref}\ R \vdash e_2 : \tau}{\Gamma; \Delta, x : \mathsf{Owned}\langle R\rangle \vdash \mathsf{borrow}(x\ \textbf{as}\ y,\ e_2) : \tau \otimes \mathsf{Owned}\langle R\rangle}\ \textit{T-Borrow}$$

$$\frac{\Gamma; \Delta_0 \vdash e : \mathsf{Bool} \qquad \Gamma; \Delta \vdash e_1 : \tau \qquad \Gamma; \Delta \vdash e_2 : \tau}{\Gamma; \Delta_0, \Delta \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau}\ \textit{T-If}$$

Note that T-Abs-Un requires the linear context to be empty ($\cdot$): an unrestricted function $\tau_1 \to \tau_2$ may be duplicated, so it must not close over linear resources—otherwise, applying the closure twice would duplicate those resources, violating linearity.

Note that T-If requires both branches to consume the same *linear context Δ. This is the key rule that prevents conditional resource leaks.*

B Extended Examples

B.1 Database Connection Pool

```
resource DbConnection

type PoolMsg =
  | Checkout(Reply[Owned[DbConnection]])
  | Return(Owned[DbConnection])
  | Shutdown
```

```

fn pool_process(conns: List[Owned[DbConnection]],
                waiting: List[Reply[Owned[DbConnection]]])
    → Never with Io, Process[PoolMsg] =
  match Process.receive() with
  | Checkout(reply) →
      match conns with
      | [conn, ..rest] →
          Reply.send(reply, conn)
          pool_process(rest, waiting)
      | [] →
          pool_process(conns, List.append(waiting, [reply]))
  | Return(conn) →
      match waiting with
      | [waiter, ..rest] →
          Reply.send(waiter, conn)
          pool_process(conns, rest)
      | [] →
          pool_process([conn, ..conns], waiting)
  | Shutdown →
      List.each(conns, fn conn →Db.close(conn))
      Process.exit(Normal)
```

Listing 19: A resource-safe connection pool

B.2 State Effect with Linear Resources

```
-- Explicit mutable state via effect
fn accumulate(items: List[Int]) →Int with State[Int] =
  List.each(items, fn x →State.modify(fn acc →acc + x))
  State.get()

-- Resource layer: mutable buffer (ownership tracked)
fn fill_buffer(data: Bytes) →Bytes with Io =
  let buf = Buffer.alloc(1024)
  let buf = Buffer.write(buf, 0, data)
  Buffer.freeze(buf)

-- Combining pure, state, and resource layers
fn process_and_store(items: List[Int], path: Path)
    → Result[Unit, IoError] with Io =
  let total = State.run(0, fn →accumulate(items))
  let file = File.open(path, Write)?
  File.write(file, Int.to_string(total))
  File.close(file)
  Ok(())
```

Listing 20: Composing effects with linear resources

B.3 Safe FFI Wrapping

```
-- Foreign resource: a C library handle
resource CHandle

foreign "C" fn c_lib_open(config: CString) →Ptr[CHandle]
foreign "C" fn c_lib_process(h: Ptr[CHandle], data: Ptr[Byte], len: CInt) →CInt
foreign "C" fn c_lib_close(h: Ptr[CHandle]) →Unit

-- Safe JAPL wrapper: all ownership is explicit
fn open_handle(config: String) →Result[Owned[CHandle], Error] with Io =
  use cstr = CString.from(config)
  let ptr = unsafe c_lib_open(cstr)
  if Ptr.is_null(ptr) then Err(OpenFailed)
  else Ok(CHandle.from_raw(ptr))

fn process_data(handle: Owned[CHandle], data: Bytes)
    → (Owned[CHandle], Result[Int, Error]) with Io =
  let result = unsafe c_lib_process(
    CHandle.as_ptr(ref handle),
    Bytes.as_ptr(data),
    Bytes.length(data)
  )
  if result < 0 then (handle, Err(ProcessFailed(result)))
  else (handle, Ok(result))

fn close_handle(handle: Owned[CHandle]) →Unit with Io =
  unsafe c_lib_close(CHandle.into_raw(handle))
```

Listing 21: Safe wrapper around a C FFI resource