Failures Are Normal and Typed: A Unified Theory of Error Handling Combining Typed Recoverable Errors with Crash-and-Restart Semantics

JAPL Language Research Group
matthew@yonedaai.com

March 2026

Abstract

Error handling remains one of the most contested design decisions in programming language theory and practice. Existing approaches fragment into incompatible philosophies: exceptions (Java, Python) conflate control flow with failure signaling; error codes (C, Go) lack compositional structure; monadic result types (Rust, Haskell) demand exhaustive handling of every failure mode; and crash-and-restart semantics (Erlang/OTP) abandon fine-grained error typing in favor of process isolation. No single approach is complete.

We present Japl's *dual error model*, which unifies typed recoverable errors with process-level crash semantics into a coherent whole. The model distinguishes two fundamentally different failure modes: *domain errors*, represented as typed `Result` values that flow through an algebraic effect system, and *process failures*, which trigger process termination and supervisory restart. We formalize this distinction using sum types, Kleisli categories over the exception monad, algebraic effects, and process algebra with crash actions. We prove three key properties: *error safety* (well-typed programs cannot silently discard domain errors), *crash containment* (a process failure cannot corrupt the state of other processes), and *supervision liveness* (permanent processes are eventually restarted after crash). Through extensive comparison with Rust, Go, Erlang, Haskell, and Java, and through case studies of an HTTP server, a database connection pool, and a distributed task worker, we demonstrate that the dual model covers all practical failure modes with minimal ceremony and maximal type safety.

1 Introduction

Every program encounters failure. Files go missing, networks drop packets, users submit malformed input, hardware degrades, and bugs escape testing. How a programming language handles these failures shapes the reliability, readability, and maintainability of the software written in it.

The history of error handling in programming languages is a history of trade-offs. Each approach optimizes for certain failure modes while leaving others poorly served:

Exceptions (Java, Python, C++). Exceptions provide a mechanism for non-local control flow transfer, allowing functions deep in the call stack to signal failures to distant callers. Java's checked exceptions Gosling et al. [2014] attempt to bring static typing to this mechanism, requiring callers to either handle or propagate declared exception types. In practice, checked exceptions have been widely criticized Eckel [2003], Bloch [2008]: they create coupling between implementation details and interface signatures, encourage catching overly broad exception types, and scale poorly in the presence of higher-order functions and generics.

Error codes and values (C, Go). C's convention of returning integer error codes and Go's explicit `error` return values Pike [2012] keep failure in the value domain, making it visible at every call site. However, neither language enforces that errors are inspected. In C, a forgotten `errno` check is a silent bug. In Go, the `if err != nil` pattern introduces substantial syntactic noise and, lacking sum types, cannot express the *type* of the error with the same precision as an algebraic data type.

Result types (Rust, Haskell). Rust's `Result<T, E>` Matsakis and Klock [2014] and Haskell's `Either a b` Marlow [2010] embed failure in the type system as sum types. The compiler enforces that callers handle both success and failure cases. Rust's `?` operator provides syntactic sugar for propagation. This approach is compositionally elegant but can be onerous: converting between error types, chaining fallible operations, and handling errors at appropriate levels of abstraction all require careful design. More fundamentally, Rust lacks a clean answer for truly unexpected failures—`panic!` exists but is designed as a last resort, not a systematic recovery mechanism.

Crash-and-restart semantics (Erlang/OTP). Erlang takes a radically different approach: rather than preventing or carefully handling every failure, it embraces failure as normal Armstrong [2003]. Processes are lightweight and isolated. When a process encounters an unexpected condition, it crashes. A *supervisor* process detects the crash and restarts the failed process with fresh state. This "let it crash" philosophy Armstrong [2007] has proven remarkably effective for building fault-tolerant telecommunications and distributed systems. However, Erlang's dynamic typing means that error types are not checked at compile time, and the boundary between errors that should be handled locally and errors that should trigger a crash is left to programmer convention.

The gap. No existing language satisfactorily unifies these approaches. Rust handles domain errors beautifully but has no process-level crash recovery. Erlang handles unexpected failures gracefully but provides no compile-time error typing. Go has neither sum types nor supervision. Java's checked exceptions were the right instinct—making failure visible in types—executed with the wrong mechanism.

JAPL addresses this gap with a *dual error model* that cleanly separates two fundamentally different kinds of failure:

1. **Domain errors** (`Result` types + `Fail` effect): Expected, recoverable failures such as parse errors, validation failures, and not-found conditions. These are values, tracked by the type system and the effect system, that must be handled by the caller.

2. **Process failures** (crash + supervision): Unexpected failures such as invariant violations, corrupted state, and unrecoverable resource loss. These terminate the enclosing process, and a supervisor restarts it with fresh state.

This paper formalizes the dual model, proves its key safety and liveness properties, and demonstrates its practical advantages through comparison and case study.

Contributions.

- A formal framework for dual-mode error handling combining sum types, algebraic effects, and process algebra (Section 3).
- The design and specification of JAPL's dual error model, including the `Fail` effect and its integration with supervision (Sections 4 to 7).
- Proofs of error safety, crash containment, and supervision liveness (Section 11).
- Comprehensive comparison with five major error handling paradigms (Section 8).
- Implementation strategies and performance analysis (Section 9).
- Three detailed case studies demonstrating the model in practice (Section 10).

2 Background and Related Work

2.1 Exception Safety and the C++ Legacy

Stroustrup introduced exceptions to C++ as a mechanism for signaling errors across abstraction boundaries Stroustrup [1994]. The subsequent development of *exception safety* guarantees—basic, strong, and no-throw—by Abrahams Abrahams [2000] and Sutter Sutter [1999] revealed the deep interaction between exceptions and resource management. The RAII (Resource Acquisition Is Initialization) pattern emerged as the C++ community's solution to ensuring resource cleanup in the presence of exceptions, a concern that JAPL addresses through ownership types and linear resource management.

2.2 Monadic Error Handling

Moggi's seminal work on monads as a structuring mechanism for computational effects Moggi [1991] provided the theoretical foundation for treating errors as first-class values. Wadler Wadler [1992, 1995] popularized the application of monads in functional programming, showing how the *exception monad* $T(A) = A + E$ (for a fixed error type E) could structure error-handling code. Haskell's `Either` monad and its `MonadError` typeclass Jones [1995] realize this approach, enabling compositional error handling through `do`-notation.

However, the monadic approach in Haskell suffers from practical limitations. The "monad transformer" stack needed to combine multiple effects (errors, state, IO) introduces complexity and performance overhead Kiselyov et al. [2013]. More fundamentally, Haskell also has a separate, dynamically-typed exception system for IO errors Marlow et al. [2001], creating a dualism that is acknowledged but not cleanly resolved.

2.3 Rust's Result and Panic

Rust Matsakis and Klock [2014], Klabnik and Nichols [2019] made the deliberate decision to separate *recoverable* errors (represented by `Result<T, E>`) from *unrecoverable* errors (represented by `panic!`). The `?` operator provides ergonomic error propagation. The `From` trait enables automatic error type conversion during propagation.

Rust's approach is the closest antecedent to Japl's dual model. However, Rust lacks several features that Japl provides: (1) Rust has no built-in lightweight process model, so `panic!` typically aborts the entire program or unwinds a single thread—there is no supervision. Libraries such as `tokio` (asynchronous task runtime) and `rayon` (data-parallel work-stealing) provide lightweight tasks, but these are library abstractions that lack the isolation and crash-containment primitives of a language runtime: a panic in a `tokio` task can poison shared state, and neither library provides supervision trees or typed crash reasons; (2) Rust's error handling is not integrated with an effect system, so the compiler does not track which functions can fail with which error types as a composable annotation; (3) Rust's `panic!` is designed as a last resort, not as a deliberate recovery strategy.

2.4 Go's Error Values

Go Pike [2012], Donovan and Kernighan [2015] represents errors as values implementing the `error` interface. Functions return `(T, error)` tuples. This approach is explicit but lacks algebraic structure: Go has no sum types, so error handling cannot benefit from exhaustive pattern matching. The `if err != nil` pattern is pervasive and syntactically heavy. Go's `panic`/`recover` mechanism exists but is discouraged for normal error handling.

Recent additions to Go's error handling—`errors.Is`, `errors.As`, and error wrapping with `fmt.Errorf("%w", err)`—are runtime-typed approximations of what algebraic data types provide at compile time.

2.5 Erlang's "Let It Crash" Philosophy

Armstrong's doctoral thesis Armstrong [2003] formalized the philosophy that has made Erlang/OTP the gold standard for fault-tolerant systems. The key insights are:

1. **Process isolation**: Processes share no memory. A bug in one process cannot corrupt another.

2. **Crash as recovery**: Rather than attempting to handle every conceivable failure, a process that encounters unexpected state simply terminates. A supervisor detects this and restarts the process with known-good initial state.

3. **Error kernel pattern**: The system is structured so that a small, well-tested "error kernel" of supervisor processes manages the lifecycle of a larger population of worker processes that may fail Armstrong [2007].

4. **Supervision trees**: Supervisors are organized hierarchically, with strategies (one-for-one, all-for-one, rest-for-one) governing how failures propagate and how restarts occur Logan et al. [2010].

Erlang's limitation is the absence of static typing for errors. A process may crash with any term as the "reason," and supervisors make restart decisions based on runtime pattern matching against crash reasons. This works in practice but leaves significant error-handling logic invisible to static analysis.

2.6 Algebraic Effects and Handlers

Plotkin and Pretnar's algebraic effects and handlers Plotkin and Pretnar [2009, 2013] provide a principled framework for structuring computational effects. An algebraic effect declares a set of operations; a handler provides interpretations for those operations. Leijen Leijen [2017] demonstrated practical effect systems with row-polymorphic effect typing in the Koka language.

JAPL's `Fail` effect is an instance of the exception effect in this framework. The algebraic effects perspective is crucial because it makes error-handling capability *compositional*: functions declare which error effects they require, and handlers can interpret these effects at appropriate boundaries.

2.7 Failure Detectors

Chandra and Toueg Chandra and Toueg [1996] formalized the notion of failure detectors in distributed systems, classifying them by completeness and accuracy properties. This work is relevant to JAPL's supervision model: a supervisor acts as a local failure detector for its children, and the completeness guarantee (every crashed process is eventually detected) is essential for the supervision liveness property we prove in Section 11.

2.8 Process Algebras

The π-calculus Milner [1999] and CSP Hoare [1985] provide formal foundations for reasoning about concurrent processes and message passing. We extend these frameworks with explicit crash and restart actions to model JAPL's supervision semantics.

3 Formal Framework

We develop a formal framework that unifies typed error handling with crash semantics. The framework has three components: (1) an algebraic treatment of error types as sum types, (2) categorical semantics of failure using Kleisli categories, and (3) a process algebra extended with crash actions.

3.1 Error Types as Sum Types

Definition 3.1 (Error type)**.** *An* error type E *is an algebraic data type (sum type) whose constructors represent distinct failure modes:*

$$E = C_1(\tau_1) \mid C_2(\tau_2) \mid \cdots \mid C_n(\tau_n)$$

where each C_i is a constructor carrying data of type τ_i.

Definition 3.2 (Result type)**.** *The* result type $\texttt{Result}\langle A, E\rangle$ *is a binary sum type:*

$$\texttt{Result}\langle A, E\rangle = \mathsf{Ok}(A) + \mathsf{Err}(E)$$

The result type forms a *monad*, which we call the exception monad for error type E.

Definition 3.3 (Exception monad)**.** *For a fixed error type E, the exception monad T_E is defined by:*

$$\begin{aligned} T_E(A) &= A + E \\ \eta_A(a) &= \mathsf{Ok}(a) \\ \mu_A(\mathsf{Ok}(\mathsf{Ok}(a))) &= \mathsf{Ok}(a) \\ \mu_A(\mathsf{Ok}(\mathsf{Err}(e))) &= \mathsf{Err}(e) \\ \mu_A(\mathsf{Err}(e)) &= \mathsf{Err}(e) \end{aligned}$$

The monadic bind (Kleisli composition) is:

$$m \gg= f = \begin{cases} f(a) & \text{if } m = \mathsf{Ok}(a) \\ \mathsf{Err}(e) & \text{if } m = \mathsf{Err}(e) \end{cases}$$

Proposition 3.4 (Monad laws)**.** *T_E satisfies the monad laws:*

1. *Left identity: $\eta(a) \gg= f = f(a)$*
2. *Right identity: $m \gg= \eta = m$*
3. *Associativity: $(m \gg= f) \gg= g = m \gg= (\lambda a.\, f(a) \gg= g)$*

Proof. By case analysis on the structure of m. For (1): $\eta(a) \gg= f = \mathsf{Ok}(a) \gg= f = f(a)$. For (2): if $m = \mathsf{Ok}(a)$, then $m \gg= \eta = \eta(a) = \mathsf{Ok}(a) = m$; if $m = \mathsf{Err}(e)$, then $m \gg= \eta = \mathsf{Err}(e) = m$. For (3): the proof proceeds by case analysis on m: if $m = \mathsf{Err}(e)$, both sides reduce to $\mathsf{Err}(e)$; if $m = \mathsf{Ok}(a)$, both sides reduce to $f(a) \gg= g$. $\square$

3.2 Kleisli Categories and Partial Functions

The categorical semantics of error handling can be understood through the Kleisli category of the exception monad.

Definition 3.5 (Kleisli category)**.** *The Kleisli category* $\mathbf{Kl}(T_E)$ *for the exception monad* T_E *has:*

- *Objects: types* $A, B, C, \ldots$
- *Morphisms* $A \to B$*: functions* $A \to T_E(B) = A \to (B + E)$
- *Identity:* $\eta_A : A \to A + E$
- *Composition:* $g \circ_K f = \lambda a.\, f(a) \gg\!= g$

The Kleisli category $\mathbf{Kl}(T_E)$ is isomorphic to a category of *partial functions* with typed partiality. A morphism $f : A \to B + E$ in $\mathbf{Kl}(T_E)$ is a function that either produces a B or fails with a reason from E. This gives us a precise categorical characterization of Japl's fallible functions.

Proposition 3.6. *Kleisli composition of fallible functions is associative and preserves error propagation: if* $f : A \to B + E$ *fails, then* $g \circ_K f$ *fails with the same error, regardless of* g.

This property corresponds directly to Japl's `?` operator semantics: error propagation is short-circuiting.

3.3 Algebraic Effects for Errors

We model Japl's `Fail` effect using the framework of algebraic effects and handlers Plotkin and Pretnar [2009].

Definition 3.7 (Fail effect)**.** *The* $\texttt{Fail}\langle E\rangle$ *effect is an algebraic effect with a single operation:*

$$\mathsf{fail} : E \to \bot$$

The operation $\mathsf{fail}(e)$ *signals a failure with reason* $e : E$ *and does not return (indicated by the return type* $\bot$*).*

Definition 3.8 (Fail handler)**.** *A handler for* $\texttt{Fail}\langle E\rangle$ *interprets the effect, converting an effectful computation into a* `Result` *value:*

$$\mathsf{catch} : (\forall \alpha.\, (\texttt{Fail}\langle E\rangle \Rightarrow \alpha) \to \texttt{Result}\langle \alpha, E\rangle)$$

Specifically:

$$\mathsf{catch}(\mathsf{return}\ v) = \mathsf{Ok}(v)$$
$$\mathsf{catch}(\mathsf{fail}\ e\ k) = \mathsf{Err}(e)$$

where k *is the (discarded) continuation.*

The algebraic effects framework provides several advantages over the monadic approach:

1. **Composition**: Multiple `Fail` effects with different error types compose naturally via row-polymorphic effect rows, without the need for monad transformers.
2. **Separation of concern**: The code that *raises* an error and the code that *handles* it are decoupled; the handler can be installed at any point in the call chain.
3. **Compiler tracking**: The effect system tracks which functions may fail and with which error types, enabling static analysis.

3.4 Process Algebra with Crash Actions

To model process failures and supervision, we extend a process algebra with explicit crash and restart actions.

Definition 3.9 (Extended process algebra)**.** *A process* P *is defined by the grammar:*

$$\begin{array}{lll} P ::= & \mathbf{0} & \textit{(terminated)} \\ & \mid \alpha.P & \textit{(action prefix)} \\ & \mid P_1 \parallel P_2 & \textit{(parallel composition)} \\ & \mid P_1 + P_2 & \textit{(choice)} \\ & \mid \mathsf{crash}(r) & \textit{(crash with reason } r\textit{)} \\ & \mid \mathsf{sup}(S, [P_1, \ldots, P_n]) & \textit{(supervision)} \\ & \mid\ !P & \textit{(replication)} \end{array}$$

where α *ranges over actions (send, receive, internal),* r *is a crash reason, and* S *is a supervision strategy.*

Definition 3.10 (Crash transition)**.** *A process may transition to a crashed state:*

$$\frac{}{\mathsf{crash}(r) \xrightarrow{\mathsf{crash}(r)} \mathbf{0}}$$

Definition 3.11 (Supervision transition)**.** *A supervisor observes a child crash and restarts it:*

$$\frac{P_i \xrightarrow{\mathsf{crash}(r)} \mathbf{0} \quad S(i, r) = \mathsf{restart}}{\mathsf{sup}(S, [\ldots, P_i, \ldots]) \xrightarrow{\tau} \mathsf{sup}(S, [\ldots, P_i^0, \ldots])}$$

where P_i^0 *is the initial state of process* i.

The supervision strategy S determines the restart behavior:

Definition 3.12 (Supervision strategies)**.**

$$
\begin{aligned}
\mathsf{OneForOne}(i, r) &= \mathsf{restart}(i) \\
\mathsf{AllForOne}(i, r) &= \mathsf{restart}(1), \ldots, \mathsf{restart}(n) \\
\mathsf{RestForOne}(i, r) &= \mathsf{restart}(i), \mathsf{restart}(i+1), \ldots, \mathsf{restart}(
\end{aligned}
$$

3.5 Connecting the Two Layers

The key insight of JAPL's dual model is that the two formalisms—Kleisli categories for domain errors and process algebra for crashes—operate at different levels of abstraction with a well-defined boundary between them.

Definition 3.13 (Error/crash boundary)**.** *A boundary function β maps from the domain error level to the process level:*

$$\beta : \mathtt{Result}\langle A, E\rangle \to \mathsf{Proc}$$

such that:

$$
\begin{aligned}
\beta(\mathsf{Ok}(a)) &= \textit{continue with } a \\
\beta(\mathsf{Err}(e)) &= \textit{continue with error handling for } e \\
\textit{(invariant violation)} &\to \mathsf{crash}(r)
\end{aligned}
$$

The boundary is asymmetric: domain errors can always be converted to process actions (either handling the error or choosing to crash), but crashes cannot be converted back to domain errors within the crashed process. Recovery from crashes happens at the *supervisor* level, not within the failed process itself.

4 JAPL's Dual Error Model

JAPL provides two complementary mechanisms for handling failure, each appropriate for a different class of errors.

4.1 Recoverable Domain Errors: Result Types

Domain errors represent expected failure modes—conditions that are part of the normal operation of a system. A database query may not find a matching row. A JSON parser may encounter malformed input. A user may submit an invalid form. These are not bugs; they are anticipated outcomes that the program must handle.

In JAPL, domain errors are represented as typed sum types and propagated through the `Fail` effect:

```
type AppError =
  | NotFound
  | Unauthorized
  | InvalidInput(String)
  | DbError(String)

fn get_user(id: UserId) →User
    with Io, Fail[AppError] =
  let row = db_find_user(id)?
  decode_user(row)?
```

Several design properties are notable:

Exhaustive handling. Because error types are algebraic data types, pattern matching on errors is checked for exhaustiveness by the compiler:

```
fn handle_error(err: AppError) →Response =
  match err with
  | NotFound →response(404, "not found")
  | Unauthorized →response(401, "unauthorized")
  | InvalidInput(msg) →
      response(400, "bad input: " ++ msg)
  | DbError(msg) →
      log_error(msg)
      response(500, "internal error")
```

If a new variant is added to `AppError`, every match expression is flagged as incomplete until updated. This is the same exhaustiveness checking that Rust and Haskell provide, applied to error handling.

Compositional propagation. The `?` operator propagates errors through the call chain:

```
fn process_order(req: Request) →OrderResult
    with Io, Net, Fail[OrderError] =
  let user = get_user(req.user_id)?
  let items = validate_items(req.items)?
  let payment = charge_payment(user, items)?
  create_order(user, items, payment)?
```

Each `?` invocation is syntactic sugar for pattern matching on the result: if Ok, unwrap; if Err, propagate immediately. The type system ensures that the propagated error type is compatible with the function's declared `Fail` effect.

Error type composition. When a function calls subfunctions with different error types, conversion is required. The idiomatic JAPL approach uses a `handle` block, which leverages the effect system to intercept and re-raise with a different error type:

```
fn get_user_profile(id: UserId) →Profile
    with Io, Fail[AppError] =
  let user = get_user(id)?
  let prefs = handle get_preferences(id) with
    | Fail(e) →Fail.raise(DbError(show(e)))
  { user, preferences = prefs }
```

The `handle` block intercepts the `Fail` effect from `get_preferences` and re-raises it as an `AppError`, keeping the conversion visible and explicit. For simple cases, the pipeline form `|> map_err(fn e -> DbError(show(e)))?` is also available and resembles Rust's `From` trait. In both cases, JAPL's effect system tracks the error type as part of the function's signature.

4.2 Process Failures: Crash and Restart

Process failures represent unexpected conditions—situations where the process's internal state may be corrupted, where an invariant has been violated, or where a resource has been irreparably lost. The appropriate response is not to attempt recovery within the process, but to terminate the process and let a supervisor start a fresh instance.

```
fn critical_worker(state: State)
   → Never with Process[WorkerMsg] =
  match Process.receive() with
  | ProcessTask(task) →
      -- If the invariant is violated, the
      -- process crashes. The supervisor
      -- restarts it with fresh state.
      assert valid_invariant(state)
      let new_state = handle_task(state, task)
      critical_worker(new_state)
  | Shutdown →
      cleanup(state)
      Process.exit(Normal)
```

Key properties of crash semantics in JAPL:

Process isolation. Processes do not share mutable state. A crash in one process cannot corrupt the memory of another. This is enforced by the language's type system: values sent between processes must be immutable (and therefore safely shareable) or ownership must be explicitly transferred.

Typed crash reasons. Unlike Erlang, where a process can crash with any term, JAPL provides structured crash reasons:

```
type CrashReason =
  | Normal
  | AssertionFailed(String, Location)
  | ResourceExhausted(String)
  | InvariantViolation(String)
  | Timeout
  | Custom(String)
```

This allows supervisors to make informed restart decisions based on the crash reason's type, not just runtime string matching.

Deterministic cleanup. When a process crashes, all its owned resources (file handles, network connections, etc.) are deterministically released through JAPL's ownership system. The linear type system guarantees that every resource is consumed exactly once, even on the crash path.

4.3 The Complementarity Principle

The two error modes are complementary, not overlapping. They address different questions:

	Domain Errors	**Process Failures**
Nature	Expected	Unexpected
Mechanism	`Result` + `Fail`	`crash` + supervision
Scope	Function/call chain	Process
Recovery	Caller handles	Supervisor restarts
State	Preserved	Discarded
Examples	Not found, invalid input, timeout	Corrupted state, hardware fault, bug

A well-designed JAPL application uses domain errors for all anticipated failure modes and reserves crashes for conditions where the process's state can no longer be trusted.

5 The Error/Crash Boundary

The most important design decision in JAPL's error model is determining when to use `Result` types and when to crash. This section provides principled guidance.

5.1 The Decision Criterion

Definition 5.1 (Error/Crash Criterion)**.** *A failure should be handled as a domain error (`Result`) if and only if:*

1. *The failure is* anticipated*: it represents a known failure mode of the operation.*
2. *The caller can* meaningfully respond*: there exists a sensible action the caller can take.*
3. *The process state remains* consistent*: the failure does not indicate state corruption.*

If any of these conditions is not met, the failure should trigger a process crash.

The following decision table summarizes the criterion for common failure scenarios:

Failure scenario	**Anticipated?**	**Actionable?**
Not-found query	✓	✓
Parse error	✓	✓
Auth rejected	✓	✓
Network timeout	✓	✓
Invalid user input	✓	✓
Assertion failure	×	×
Corrupted state	×	×
Unrecoverable resource	×	×
Index out of bounds	×	×
DB conn lost mid-txn	∼	∼
OOM in subsystem	∼	×

× Crash

Decision rule (informal). Ask: "Can the caller do something sensible with this failure, and is my state still trustworthy?" If yes, return a `Result`. If no, crash and let the supervisor restore a known-good state.

Example 5.2 (Database query). *A "not found" result from a database query satisfies all three criteria: it is anticipated (queries may not match), the caller can respond (return a 404, use a default value, etc.), and the state is consistent (the database connection is fine). This is a domain error.*

A database *connection loss* during a transaction *may violate condition (3): if the process was tracking transaction state, that state is now inconsistent with the database. The appropriate response is to crash and let the supervisor restart with a fresh connection.*

5.2 The Error Kernel Pattern

Armstrong's *error kernel* pattern Armstrong [2003] structures a system so that a small, well-tested core manages the lifecycle of larger, more complex subsystems:

```
-- The error kernel: simple, well-tested,
-- rarely crashes
fn app_supervisor() →Never
    with Process[SupervisorMsg] =
  Supervisor.start(
    strategy = OneForOne,
    max_restarts = 10,
    max_seconds = 60,
    children = [
      { id = "db_pool"
      , start = fn →db_pool_supervisor()
      , restart = Permanent
      , shutdown = Timeout(5000)
      },
      { id = "web"
      , start = fn →web_supervisor()
      , restart = Permanent
      , shutdown = Timeout(10000)
      },
    ]
  )

-- Worker processes: may crash, will be
-- restarted by supervisors
fn request_handler(conn: TcpConn) →Never
    with Process[HttpMsg], Io =
  let req = Http.parse_request(conn)
  let resp = match route(req) with
    | Ok(handler) →
        match handler(req) with
        | Ok(resp) →resp
        | Err(e) →error_response(e)
    | Err(_) →response(404, "not found")
  Http.send_response(conn, resp)
  request_handler(conn)
```

In this architecture:

- The `app_supervisor` and `db_pool_supervisor` form the error kernel. They are simple, well-tested, and their only job is to supervise children.
- The `request_handler` processes do the complex work. If one crashes (due to a bug, a malformed request that bypasses validation, or a resource issue), the supervisor restarts it.
- Within each request handler, *domain errors* are handled with `Result` types: routing failures return 404, handler errors return appropriate HTTP status codes.

5.3 Design Principles for the Boundary

We articulate five principles for managing the error/crash boundary:

1. **Domain errors at API boundaries.** Public-facing APIs (HTTP endpoints, library functions, message handlers) should return `Result` types so callers can respond appropriately.

2. **Crashes for invariant violations.** If a condition is reached that "should never happen" according to the system's design, crash. An `assert` that fails indicates a bug, not a user error.

3. **Convert at boundaries.** When calling a subsystem that may crash, wrap the call at the boundary: spawn a process, send the request, and handle the possible `ProcessDown` message as a domain error.

4. **Narrow the error kernel.** Keep the supervision hierarchy simple and the supervisor code minimal.

Complexity belongs in worker processes, which can crash safely.

5. **Escalate when uncertain.** If a process encounters a condition it does not know how to handle, crashing is safer than continuing with potentially corrupted state. The supervisor has a broader view and can make a better recovery decision.

```
-- Principle 3: Converting crashes to
-- domain errors at a boundary
fn safe_compute(input: Data) →Result[Output,
     ComputeError]
   with Process, Io =
  let worker = Process.spawn(fn →
    let result = dangerous_computation(input)
    Process.send(Process.self_parent(), Done(result))
  )
  Process.monitor(worker)
  match Process.receive_with_timeout(5000) with
  | Done(result) →Ok(result)
  | ProcessDown(^worker, reason) →
     Err(ComputeFailed(show(reason)))
  | _ →Err(ComputeTimeout)
```

6 Effect System Integration

JAPL's effect system is central to its error handling model. The `Fail` effect makes failure a tracked, composable property of function signatures.

6.1 The Fail Effect

The `Fail[E]` effect indicates that a function may fail with an error of type E. It is one of JAPL's built-in effects:

Effect	Meaning
`Pure`	No effects (identity)
`Io`	File system, console, clock
`Net`	Network access
`State[s]`	Mutable state of type s
`Process`	Process operations
`Fail[e]`	May fail with error type e
`Async`	Asynchronous operations

The `Fail[E]` effect is parameterized by the error type E, enabling the compiler to track not just *that* a function can fail, but *how* it can fail.

6.2 Effect Composition

Effects in JAPL form a commutative, idempotent monoid under union:

$$\begin{aligned} \mathcal{E}_1 \cup \mathcal{E}_2 &= \mathcal{E}_2 \cup \mathcal{E}_1 && \text{(commutativity)} \\ \mathcal{E} \cup \mathcal{E} &= \mathcal{E} && \text{(idempotency)} \\ \mathcal{E} \cup \texttt{Pure} &= \mathcal{E} && \text{(identity)} \end{aligned}$$

When a function calls subfunctions, its effect set is the union of all callee effects:

```
-- Effect inference: the compiler computes
-- the union of all effects in the body
fn process_request(req: Request) →Response
   with Io, Net, Fail[AppError] =
  -- read_config: Io, Fail[ConfigError]
  let config = read_config("/etc/app.conf")
    ▷ map_err(to_app_error)?
  -- http_fetch: Net
  let data = Http.get(config.api_url)?
  -- parse_response: Fail[ParseError]
  let parsed = parse_response(data)
    ▷ map_err(to_app_error)?
  to_response(parsed)
```

The declared effects `Io, Net, Fail[AppError]` must be a superset of the inferred effects. If the programmer omits a required effect, the compiler reports an error.

6.3 Multiple Fail Effects

A function may have multiple `Fail` effects with different error types. This is supported through effect row polymorphism:

```
fn validate_and_store(input: RawInput)
    → StoredData
    with Io, Fail[ValidationError],
         Fail[StorageError] =
  let validated = validate(input)?
    -- raises Fail[ValidationError]
  store(validated)?
    -- raises Fail[StorageError]
```

The caller must handle (or propagate) both error types. This is more precise than a single unified error type, enabling callers to handle different failure modes differently:

```
fn handle_input(input: RawInput)
    → Response with Io =
  match Fail.catch(fn →
    Fail.catch(fn →
      validate_and_store(input)
    )
  ) with
  | Ok(Ok(data)) →json(200, data)
  | Ok(Err(storage_err)) →
      log_error(storage_err)
      response(503, "storage unavailable")
  | Err(validation_err) →
      response(400, show(validation_err))
```

Alternatively, the caller can unify the error types:

```
type InputError =
  | Validation(ValidationError)
  | Storage(StorageError)

fn handle_input_unified(input: RawInput)
    → Response with Io =
  let result = Fail.catch(fn →
    validate_and_store(input)
      ▷ map_fail(Validation)
      ▷ map_fail(Storage)
  )
  match result with
  | Ok(data) →json(200, data)
  | Err(Validation(e)) →response(400, show(e))
  | Err(Storage(e)) →response(503, show(e))
```

6.4 Effect Handlers as Error Boundaries

The `Fail.catch` handler serves as an error boundary, converting an effectful computation into a pure `Result` value. This is the primary mechanism for discharging the `Fail` effect:

```
-- Type: Fail.catch :
--   (fn() →a with Fail[e]) →Result[a, e]
fn main() →Unit with Io =
  let result = Fail.catch(fn →
    complex_fallible_computation()
  )
  match result with
  | Ok(value) →
      Io.println("Success: " ++ show(value))
  | Err(e) →
      Io.println("Error: " ++ show(e))
```

The handler eliminates the `Fail` effect from the computation's effect set, converting it into a `Result` value in a context with strictly fewer effects. This gives precise control over where in the call chain errors are handled.

6.5 Compiler Tracking

The `Fail` effect enables several static analyses:

1. **Unhandled errors**: If a function's effect set includes `Fail[E]` but no handler discharges it, the compiler requires the caller to either handle or propagate the effect.

2. **Dead error branches**: If an error variant is never constructed by any reachable code path, the compiler can warn that a match branch is unreachable.

3. **Error documentation**: The effect signature serves as machine-checked documentation of all failure modes, visible in IDEs, generated documentation, and type signatures.

4. **Error compatibility**: When composing modules, the compiler checks that error types are compatible across module boundaries.

7 Supervision as Error Recovery

While domain errors are handled within function call chains, process failures are handled by the supervision hierarchy. Supervision is JAPL's mechanism for systematic recovery from unexpected failures.

7.1 Supervision Trees

A supervision tree is a hierarchical organization of processes where each non-leaf node is a supervisor that monitors and manages its children:

```
fn start_application() →Pid[SupervisorMsg]
    with Process =
  Supervisor.start(
    strategy = OneForOne,
    max_restarts = 5,
    max_seconds = 60,
    children = [
      { id = "db_pool"
      , start = fn →DbPool.start(db_config)
      , restart = Permanent
      , shutdown = Timeout(5000)
      },
      { id = "cache"
      , start = fn →Cache.start(cache_config)
      , restart = Permanent
      , shutdown = Timeout(3000)
      },
      { id = "web_server"
      , start = fn →WebServer.start(web_config)
      , restart = Permanent
      , shutdown = Timeout(10000)
      },
    ]
  )
```

7.2 Restart Strategies

JAPL provides three restart strategies, following Erlang/OTP:

OneForOne. Only the failed child is restarted. This is appropriate when children are independent.

AllForOne. All children are restarted when any one fails. This is appropriate when children are interdependent and their states must be consistent.

RestForOne. The failed child and all children started after it are restarted. This is appropriate when later children depend on earlier ones.

7.3 Typed Crash Reasons

JAPL's typed crash reasons enable supervisors to make informed decisions:

```
fn custom_supervisor(children: List[ChildSpec])
    → Never with Process[SupervisorMsg] =
  match Process.receive() with
  | ChildCrashed(child_id, reason) →
    match reason with
    | Normal →
        -- Normal exit, do not restart
        custom_supervisor(
          remove_child(children, child_id))
    | AssertionFailed(_, _) →
        -- Bug detected, restart with
        -- fresh state
        restart_child(children, child_id)
        custom_supervisor(children)
    | ResourceExhausted(_) →
        -- Resource issue, wait before
        -- restarting
        Process.sleep(1000)
        restart_child(children, child_id)
        custom_supervisor(children)
    | InvariantViolation(_) →
        -- Serious issue, escalate
        log_critical(child_id, reason)
        Process.exit(reason)
```

This is a significant improvement over Erlang, where crash reasons are untyped terms. The compiler ensures that the supervisor handles all crash reason variants.

7.4 Restart Limits

Supervisors enforce restart limits to prevent infinite restart loops:

```
-- If more than max_restarts occur within
-- max_seconds, the supervisor itself crashes,
-- escalating to its parent supervisor.
Supervisor.start(
  strategy = OneForOne,
  max_restarts = 5,
  max_seconds = 60,
  children = [...]
)
```

This creates a natural escalation mechanism: if a subsystem cannot recover by restarting, the failure propagates upward through the supervision tree until a higher-level supervisor can respond (perhaps by restarting the entire subsystem with different configuration, or by shutting down gracefully).

7.5 Child Specifications

Each child process is described by a specification that governs its lifecycle:

Restart type	Meaning
`Permanent`	Always restarted
`Transient`	Restarted only on abnormal exit
`Temporary`	Never restarted

Shutdown type	Meaning
`Timeout(ms)`	Send shutdown signal, wait up to *ms*
`Brutal`	Terminate immediately

The combination of restart type, shutdown type, and crash reason gives supervisors fine-grained control over the recovery process.

7.6 Supervision and the Effect System

Supervision interacts with the effect system through the `Process` effect. Functions that spawn, monitor, or communicate with processes must declare the `Process` effect. This makes the presence of concurrent, potentially-failing subcomputations visible in the type signature:

```
-- The Process effect signals that this
-- function participates in the process model
fn start_worker_pool(config: PoolConfig)
    → WorkerPool
    with Process, Io =
  let sup = Supervisor.start(
    strategy = OneForOne,
    max_restarts = config.max_restarts,
    max_seconds = config.max_seconds,
    children = List.map(
      List.range(1, config.pool_size),
      fn i →{
        id = "worker-" ++ Int.to_string(i),
        start = fn →worker(config.worker_config),
        restart = Permanent,
        shutdown = Timeout(5000),
      }
    )
  )
  { supervisor = sup, config }
```

8 Comparison with Existing Approaches

We now compare JAPL's dual error model with the error handling approaches of five major languages.

8.1 Rust: Result Without Supervision

Rust's `Result<T, E>` is the direct inspiration for JAPL's domain error handling. The comparison reveals what Rust lacks:

Feature	**Rust**	**Japl**
Typed recoverable errors	✓	✓
`?` propagation	✓	✓
Exhaustive matching	✓	✓
Effect tracking for errors	×	✓
Lightweight processes	×	✓
Supervision trees	×	✓
Typed crash reasons	×	✓
Crash → restart recovery	×	✓

Rust's `panic!` unwinds the thread stack and either aborts the program or is caught by `std::panic::catch_unwind`, which is explicitly documented as not a general-purpose error recovery mechanism. There is no equivalent of supervision—a panicked thread is simply gone.

In Japl, the equivalent of Rust's `Result` handles the same domain errors with equal type safety. The difference is that Japl also provides a systematic answer for Rust's `panic!` cases: crash the process, let the supervisor restart it.

8.2 Go: Error Values Without Structure

Go's approach is characterized by simplicity and explicitness, but lacks algebraic structure:

Feature	**Go**	**Japl**
Errors as values	✓	✓
Sum types for errors	×	✓
Exhaustive matching	×	✓
Propagation operator	×	✓
Effect tracking	×	✓
Error type composition	×	✓
Process supervision	×	✓

Go's error handling suffers from two key weaknesses that Japl addresses: (1) the `if err != nil` boilerplate, which Japl eliminates with the `?` operator, and (2) the inability to express error types precisely, which Japl solves with algebraic data types and the `Fail` effect.

Go does have `goroutines`, which are lightweight like Japl processes, but Go provides no built-in supervision mechanism. A crashed goroutine is simply lost, and any `panic` in a goroutine that is not recovered will crash the entire program.

8.3 Erlang: Crash Semantics Without Typed Errors

Erlang is the inspiration for Japl's process model and supervision. The comparison highlights what static typing adds:

Feature	**Erlang**	**Japl**
Lightweight processes	✓	✓
Process isolation	✓	✓
Supervision trees	✓	✓
"Let it crash"	✓	✓
Typed error values	×	✓
Exhaustive error matching	×	✓
Typed crash reasons	×	✓
Compile-time error tracking	×	✓

Erlang's dynamic typing means that error handling relies on runtime conventions. Common patterns like `{ok, Value}` and `{error, Reason}` are not enforced by the type system. Dialyzer Lindahl and Sagonas [2006] provides some static analysis but cannot match the guarantees of Japl's type system.

Japl preserves Erlang's supervision model while adding compile-time guarantees about error types, crash reasons, and error handling completeness.

8.4 Java: Checked Exceptions—Right Idea, Wrong Mechanism

Java's checked exceptions Gosling et al. [2014] share Japl's goal of making failure visible in function signatures. The execution was flawed:

Feature	**Java**	**Japl**
Failure in type signature	✓	✓
Non-local control flow	✓	×*
Composable error types	×	✓
No hidden exceptions	×	✓
Works with generics	×	✓
Works with lambdas	×	✓
Process supervision	×	✓

*Japl's `Fail` effect uses early return, not stack unwinding.

Java's checked exceptions fail with higher-order functions because Java's type system cannot express "this lambda may throw `IOException`." The result is pervasive `try-catch` blocks inside lambdas, or the use of unchecked (runtime) exceptions that bypass the checking system entirely. Japl's effect system integrates naturally with higher-order functions and generics: a function parameter can carry a `Fail` effect, and the compiler propagates it correctly.

8.5 Haskell: Pure but Complex

Haskell's approach is the most theoretically principled but also the most complex:

Feature	**Haskell**	**Japl**
Typed error values	✓	✓
Monadic composition	✓	✓
Effect tracking	Partial[*]	✓
Simple error propagation	×	✓
No hidden IO exceptions	×	✓
Process supervision	×[†]	✓
Reduced syntactic ceremony	×	✓

[*]Haskell's `IO` type bundles all effects, including exceptions, into a single opaque monad. Libraries like `polysemy` King [2019] and `fused-effects` Wu et al. [2019] provide finer-grained tracking but are not part of the standard language.

[†]Cloud Haskell Epstein et al. [2011] provides Erlang-style distribution and supervision, but it is a library, not a language feature.

Haskell's primary weakness for error handling is the coexistence of multiple, incompatible error mechanisms: `Either`, `Maybe`, `ExceptT`, IO exceptions (`throwIO`/`catch`), pure exceptions (`error`/`undefined`), and asynchronous exceptions. A Haskell programmer must understand all of these and choose appropriately. Japl provides exactly two mechanisms—`Result` and crash—with clear guidance on when to use each.

8.6 Summary

	Typed errors	Exhaustive	Effect tracked	Supervision	Typed crashes
Rust	✓	✓	×	×	×
Go	∼	×	×	×	×
Erlang	×	×	×	✓	×
Java	✓	×	∼	×	×
Haskell	✓	✓	∼	×	×
Japl	✓	✓	✓	✓	✓

Japl is the only language that achieves all five properties: typed errors, exhaustive matching, effect-tracked failures, supervision trees, and typed crash reasons.

9 Implementation

Implementing the dual error model requires careful runtime design. This section discusses the key implementation considerations.

9.1 Domain Error Implementation

Domain errors (`Result` types) are implemented as tagged unions in memory. A `Result[A, E]` value occupies $1 + \max(|A|, |E|)$ words: one tag word plus space for the larger of the two payloads. This is the standard algebraic data type representation, identical to Rust's approach.

The `?` operator compiles to a conditional branch:

```
-- Source
let x = fallible_call()?

-- Desugared (conceptual)
let x = match fallible_call() with
  | Ok(v) →v
  | Err(e) →return Err(e)
```

This is a local transformation with no runtime overhead beyond the branch. There is no stack unwinding, no exception table lookup, and no dynamic dispatch. The cost of error propagation is exactly one conditional branch per `?` invocation.

9.2 Process Failure Implementation

Process crashes are implemented through a combination of mechanisms:

Assertion failures and panics. When an `assert` fails or a `panic` is invoked, the runtime:

1. Captures the crash reason (a structured `CrashReason` value).
2. Unwinds the process's call stack, running cleanup for any owned resources (similar to C++ RAII or Rust drop).
3. Marks the process as terminated with the given crash reason.
4. Notifies all monitors and linked processes.

Stack unwinding vs. process termination. Unlike C++ or Rust, where stack unwinding is the mechanism for both resource cleanup and non-local control flow, JAPL uses unwinding *only* for resource cleanup on the crash path. Domain errors never unwind the stack; they use normal return-value propagation. This simplifies the runtime: the unwinding mechanism need only handle the crash case, and it always terminates the process.

Resource cleanup. JAPL's ownership system ensures that resources are cleaned up deterministically:

- On the normal path: resources are consumed by explicit calls (`File.close`, `Buffer.freeze`).
- On the crash path: the unwinder invokes drop handlers for all live owned resources on the stack.
- The linear type system guarantees that every resource has exactly one cleanup path, whether normal or crash.

9.3 Supervision Implementation

Supervisors are implemented as ordinary JAPL processes with special runtime support:

1. **Monitoring**: The runtime maintains a table of monitor relationships. When a process terminates, the runtime sends a `ProcessDown` message to all monitors.
2. **Restart logic**: The supervisor process receives `ProcessDown` messages and applies the configured restart strategy. This is ordinary message-handling code, not special runtime machinery.
3. **Restart limits**: The supervisor tracks restart timestamps in a circular buffer. If the number of restarts in the configured time window exceeds the limit, the supervisor itself crashes, escalating to its parent.

9.4 Performance Characteristics

Operation	**Cost**	**Notes**
`Result` creation	$O(1)$	Tag + payload
`?` propagation	$O(1)$	Conditional branch
`match` on error	$O(1)$	Tag comparison
Process crash	$O(s)$	s = stack depth
Resource cleanup	$O(r)$	r = owned resources
Supervisor restart	$O(1)$	Message send
Monitor notification	$O(m)$	m = monitor count

The key performance insight is that domain errors (the common case) are zero-overhead beyond the branch prediction cost. Process crashes (the uncommon case) have a cost proportional to the amount of cleanup required, which is acceptable for an exceptional event. Supervisor restarts involve only message passing and process creation, both of which are designed to be lightweight.

9.5 Comparison with Exception-Based Implementations

Traditional exception implementations use either *table-based* or *setjmp/longjmp-based* unwinding:

- **Table-based (C++, Java)**: Zero cost on the non-exception path, but exceptions are extremely expensive (hundreds of microseconds) due to table lookup and stack unwinding.
- **setjmp/longjmp (some C)**: Moderate cost on both paths due to register saving and restoration.

JAPL's approach avoids both of these costs for domain errors. The `?` operator is a conditional branch, costing nanoseconds. Only process crashes incur unwinding cost, and since crash recovery creates a fresh process, the total recovery time is dominated by process creation, not unwinding.

10 Case Studies

We present three case studies demonstrating JAPL's dual error model in realistic scenarios.

10.1 Case Study 1: HTTP Server

An HTTP server encounters many failure modes: malformed requests, missing resources, authentication failures, database errors, and network issues. The dual model cleanly separates these:

```
type HttpError =
  | BadRequest(String)
  | NotFound
  | Unauthorized
  | Forbidden
  | InternalError(String)

type AppError =
  | Http(HttpError)
  | Db(DbError)
  | Validation(List[FieldError])

fn handle_get_user(req: Request) →Response
    with Io =
  match parse_user_id(req) with
  | Err(_) →response(400, "bad user id")
  | Ok(id) →
      match Fail.catch(fn →get_user(id)) with
      | Ok(user) →json(200, user)
      | Err(NotFound) →
          response(404, "not found")
      | Err(Unauthorized) →
          response(401, "unauthorized")
      | Err(InvalidInput(msg)) →
          response(400, msg)
      | Err(DbError(msg)) →
          log_error("db error: " ++ msg)
          response(500, "internal error")

fn get_user(id: UserId) →User
    with Io, Fail[AppError] =
  let row = db_find_user(id)?
  decode_user(row)?
```

Each HTTP request is handled in its own process. If a handler process crashes (due to a bug, not a domain error), the supervisor restarts the request-handling infrastructure:

```
fn http_listener(port: Int) →Never
    with Process, Io =
  let listener = Tcp.listen(port)
  accept_loop(listener)

fn accept_loop(listener: TcpListener) →Never
    with Process, Io =
  let conn = Tcp.accept(listener)
  -- Each connection gets its own process
  let _ = Process.spawn(fn →
    handle_connection(conn)
  )
  accept_loop(listener)

fn handle_connection(conn: TcpConn) →Unit
    with Io =
  let req = Http.parse_request(conn)
  let resp = route_and_handle(req)
  Http.send_response(conn, resp)
  Tcp.close(conn)
```

If `handle_connection` crashes, only that one connection is affected. The listener process continues accepting new connections. If the listener itself crashes, its supervisor restarts it.

10.2 Case Study 2: Database Connection Pool

A database connection pool must handle connection failures gracefully while providing a reliable interface to consumers:

```
type PoolError =
  | NoAvailableConnection
  | QueryFailed(String)
  | ConnectionLost

type PoolMsg =
  | Checkout(Reply[DbConn])
  | Return(DbConn)
  | HealthCheck

fn pool_manager(state: PoolState) →Never
    with Process[PoolMsg], Io =
  match Process.receive() with
  | Checkout(reply) →
      match find_available(state) with
      | Some(conn) →
          Reply.send(reply, Ok(conn))
          pool_manager(mark_in_use(state, conn))
      | None →
          Reply.send(reply, Err(NoAvailableConnection))
          pool_manager(state)
  | Return(conn) →
      match validate_connection(conn) with
      | Ok(_) →
          pool_manager(mark_available(state, conn))
      | Err(_) →
          -- Connection is broken, create new one
          let new_conn = Db.connect(state.config)
          pool_manager(mark_available(state, new_conn))
  | HealthCheck →
      let state = check_all_connections(state)
      pool_manager(state)
```

The pool manager handles connection validation failures as domain errors (creating replacement connections). If the pool manager itself encounters an invariant violation (e.g., the pool state becomes inconsistent), it crashes and the supervisor restarts it:

```
fn db_pool_supervisor(config: DbConfig)
    → Never with Process =
  Supervisor.start(
    strategy = OneForOne,
    max_restarts = 3,
    max_seconds = 30,
    children = [
      { id = "pool_manager"
      , start = fn →
          let conns = initialize_connections(config)
          pool_manager(initial_state(conns))
      , restart = Permanent
      , shutdown = Timeout(5000)
      },
      { id = "health_checker"
      , start = fn →health_check_loop(config)
      , restart = Permanent
      , shutdown = Timeout(1000)
      },
    ]
  )
```

10.3 Case Study 3: Distributed Task Worker

A distributed task worker receives tasks from a coordinator, executes them, and reports results. It must handle task failures, network partitions, and coordinator unavailability:

```
type TaskError =
  | InvalidTask(String)
  | ExecutionFailed(String)
  | DependencyMissing(String)

type WorkerMsg =
  | ExecuteTask(Task, Reply[TaskResult])
  | CancelTask(TaskId)
  | UpdateConfig(WorkerConfig)

fn task_worker(state: WorkerState) →Never
    with Process[WorkerMsg], Io, Net =
  match Process.receive() with
  | ExecuteTask(task, reply) →
      let result = Fail.catch(fn →
        validate_task(task)?
        fetch_dependencies(task)?
        execute(task)?
      )
      match result with
      | Ok(output) →
          Reply.send(reply, TaskSuccess(output))
      | Err(InvalidTask(msg)) →
          Reply.send(reply, TaskRejected(msg))
      | Err(ExecutionFailed(msg)) →
          Reply.send(reply, TaskFailed(msg))
      | Err(DependencyMissing(dep)) →
          Reply.send(reply, TaskPending(dep))
      task_worker(state)
  | CancelTask(id) →
      let state = cancel_if_running(state, id)
      task_worker(state)
  | UpdateConfig(config) →
      task_worker({ state | config })
```

The worker uses `Result` types for task-level failures (which are reported to the coordinator) and relies on supervision for worker-level failures (which trigger a restart):

```
fn worker_supervisor(config: WorkerConfig)
    → Never with Process =
  Supervisor.start(
    strategy = OneForOne,
    max_restarts = 10,
    max_seconds = 60,
    children = List.map(
      List.range(1, config.worker_count),
      fn i →{
        id = "worker-" ++ Int.to_string(i),
        start = fn →
          let state = WorkerState.init(config)
          task_worker(state),
        restart = Permanent,
        shutdown = Timeout(10000),
      }
    )
  )

-- Coordinator connection with reconnection
fn coordinator_link(url: String) →Never
    with Process, Io, Net =
  match Fail.catch(fn →connect_coordinator(url)) with
  | Ok(conn) →
      register_with_coordinator(conn)
      message_loop(conn)
  | Err(e) →
      log_warning("coordinator connection "
        ++ "failed: " ++ show(e))
      Process.sleep(5000)
      coordinator_link(url)
```

In this design:

- Task validation, dependency resolution, and execution failures are domain errors, reported back to the coordinator via typed `TaskResult` messages.
- Worker crashes (due to bugs or resource exhaustion) are handled by the supervisor, which restarts the worker with fresh state.
- Coordinator connectivity issues are handled by the `coordinator_link` process, which retries connection. If it fails too many times, it crashes and escalates to its supervisor.

11 Formal Properties

We now state and prove three key properties of JAPL's dual error model.

11.1 Error Safety Theorem

Theorem 11.1 (Error Safety). *In a well-typed JAPL program, every domain error is either:*

1. *Explicitly handled (matched and acted upon), or*
2. *Explicitly propagated (via ? or effect signature).*

No domain error can be silently discarded.

Proof. The proof proceeds by structural induction on the typing derivation.

Base case: A function that calls a fallible operation (one with a `Fail[E]` effect) must either:

- Use `?` to propagate the error, which requires the calling function's effect signature to include `Fail[E]` (or a supertype).
- Use `Fail.catch` to handle the error, which eliminates the `Fail[E]` effect and produces a `Result[A, E]` value.
- Use `match` on the `Result` value, which requires exhaustive coverage of both `Ok` and `Err` variants.

In all three cases, the error is either propagated or handled. The type system prevents ignoring a `Result` value: the `Result` type is not `Unit`, so assigning it to an unused binding triggers a compiler warning, and the `Fail[E]` effect prevents calling the function in a context that does not expect failure.

Inductive step: If function f calls function g with effect `Fail[E]`, then f's effect signature must include `Fail[E]` (propagation) or f must handle the effect (handling). By the inductive hypothesis, g's errors are properly handled or propagated. The composition preserves the property.

Boundary condition: At the top level (`main` or a process entry point), if `Fail[E]` remains in the effect set, the compiler reports an error: unhandled failure effect. The programmer must install a handler before the program is accepted.

Therefore, in any well-typed program, every domain error is either handled or propagated, with no possibility of silent discard. □ □

11.2 Crash Containment Theorem

Theorem 11.2 (Crash Containment). *If process P_i crashes in a system of processes $P_1 \parallel P_2 \parallel \cdots \parallel P_n$, the state of every other process P_j ($j \neq i$) is unaffected by the crash.*

Proof. The proof relies on three properties enforced by Japl's type system and runtime:

Property 1: Memory isolation. Processes do not share mutable state. Values passed between processes are either:

- Immutable values on the shared heap (which cannot be modified by any process), or
- Linearly-typed resources whose ownership is transferred (the sending process loses access).

By construction, no process can hold a mutable reference to memory owned by another process.

Property 2: Communication isolation. Processes communicate exclusively through typed message passing. Messages are values (immutable or ownership-transferred). The runtime's message delivery mechanism copies immutable values and transfers ownership atomically.

Property 3: Crash scope. When P_i crashes:

1. The runtime terminates P_i's execution.
2. All resources owned by P_i are released (deterministic cleanup).
3. Monitor notifications are sent as messages to monitoring processes.
4. No write is performed to any memory accessible by P_j.

Since P_j's state consists of its local stack, its owned resources, and its mailbox, and since the crash of P_i can only add a `ProcessDown` message to P_j's mailbox (if P_j monitors P_i), the operational state of P_j is unaffected.

The only observable effect of P_i's crash on P_j is the arrival of a `ProcessDown` message, which is a normal message that P_j processes through its regular message-handling logic. □ □

11.3 Supervision Liveness Theorem

Theorem 11.3 (Supervision Liveness). *For a process P with restart type `Permanent` supervised by a supervisor S with restart limit (m, t) (at most m restarts in t seconds): if P crashes and fewer than m restarts have occurred in the last t seconds, then P is restarted within a bounded time.*

Proof. We model the supervisor as a process S that repeatedly performs:

1. Wait for a `ProcessDown` message (blocking receive).
2. Check the restart counter against the limit.
3. If within limits, spawn a new instance of the child process.
4. If limits exceeded, crash (escalate to parent supervisor).

Crash detection: The runtime guarantees that when P terminates, a `ProcessDown` message is enqueued in S's mailbox. This follows from the completeness property of Japl's monitoring mechanism, which is implemented in the runtime scheduler.

Message delivery: The runtime guarantees that messages in a process's mailbox are eventually delivered (processes are fairly scheduled). Therefore, S will eventually receive the `ProcessDown` message.

Restart execution: Upon receiving the message and determining that restart limits are not exceeded, S invokes the child's start function, creating a new process. Process creation is an $O(1)$ runtime operation (allocate a process structure, add to the scheduler's run queue).

Bounded time: The total time from crash to restart is bounded by:

$$T_{restart} \leq T_{detect} + T_{schedule} + T_{start}$$

where T_{detect} is the monitor notification time (one scheduler tick), $T_{schedule}$ is the time until the supervisor is scheduled (bounded by the scheduler's fairness guarantee), and T_{start} is the child creation time (bounded by process allocation cost).

Under the assumption that the scheduler provides fair scheduling with bounded delay, $T_{restart}$ is bounded.

Caveat ($T_{schedule}$ under load). The bound on $T_{schedule}$ depends on the scheduler's fairness guarantee, which is sensitive to system load. Under heavy load, the number of runnable processes may cause $T_{schedule}$ to grow, becoming effectively nondeterministic from the application's perspective. In the worst case, a slow supervisor response can itself trigger cascading timeouts: if a child's clients time out while waiting for the restart, those clients may themselves fail, causing their supervisors to restart them, amplifying the load. JAPL's runtime mitigates this by giving supervisor processes elevated scheduling priority (supervisors are scheduled before workers), but the liveness bound should be understood as holding under a well-provisioned scheduler, not as an absolute real-time guarantee. Deployers should account for scheduler latency when choosing restart-limit windows. □ □

Corollary 11.4 (Escalation termination). *The escalation chain (child $\rightarrow$ supervisor $\rightarrow$ parent supervisor $\rightarrow \cdots$) terminates: either a supervisor successfully restarts the failed subtree, or the escalation reaches the root supervisor, which terminates the application.*

Proof. The supervision tree has finite depth d. At each level, a supervisor either restarts (terminating the escalation) or exceeds its restart limit and crashes (escalating). Since the tree has finite depth, escalation terminates in at most d steps. □

11.4 Composability of Error Safety

Proposition 11.5 (Compositional error safety). *If modules M_1 and M_2 independently satisfy error safety, then their composition $M_1 \circ M_2$ (where M_1 calls functions from M_2) also satisfies error safety.*

Proof. By the effect system's composition rules: if M_2 exports a function f with effect `Fail[E]`, then any function in M_1 that calls f must either handle `Fail[E]` or include it in its own effect signature. The type checker enforces this at the module boundary. Since both modules independently satisfy error safety (all their internal error handling is complete), the composition inherits this property. □ □

12 Discussion

12.1 When the Model Breaks Down

No error model is perfect. We acknowledge several limitations:

FFI boundaries and asynchronous exceptions. When JAPL calls foreign functions via FFI, the type system cannot track errors in the foreign code. Foreign functions may throw C++ exceptions, segfault, or corrupt memory in ways that violate process isolation. JAPL mitigates this through capability-gated FFI (requiring the `Unsafe` capability) and by running FFI calls in dedicated processes that can crash without affecting the rest of the system.

A subtle hazard arises when a supervisor decides to kill a child process that is currently executing inside an FFI call. The foreign code may hold external resources (file locks, GPU memory, database transactions) that the JAPL runtime cannot release through its normal ownership-based cleanup. To address this, JAPL enforces the following protocol at the unsafe FFI boundary: (1) the runtime marks the process as "in-FFI" for the duration of the foreign call, during which asynchronous kill signals are *deferred* rather than delivered immediately; (2) when the foreign call returns, the runtime checks for a pending kill signal and, if present, invokes a user-supplied `ffi_cleanup` callback before crashing the process; (3) if the foreign call does not return within a configurable timeout, the runtime forcibly terminates the process and logs a resource-leak warning—the Crash Containment theorem (Theorem 11.2) holds for JAPL-managed resources but *not* for external resources held across a stuck FFI boundary. Programmers must therefore treat long-running or resource-holding FFI calls as inherently unsafe and design their cleanup callbacks accordingly.

Divergence (non-termination). The dual error model addresses failures that produce a value (`Result`) and failures that terminate a process (crash), but a third failure mode exists: *divergence*, where a computation enters an infinite loop and produces neither a result nor a crash. A divergent process bypasses both the `Fail` effect and the crash mechanism, silently consuming resources while its supervisor receives no `ProcessDown` notification. Japl addresses this through two BEAM-inspired mechanisms. First, each process has a configurable *watchdog timer*: if a process does not yield (via message receive, explicit yield, or function return) within a configurable time bound, the runtime forces a crash with reason `Timeout`. Second, the runtime employs *reduction counting*: each function call and loop iteration consumes a "reduction credit," and when the credit is exhausted the process is preempted, giving the scheduler an opportunity to detect runaway computation. Together, these mechanisms convert divergence into a detectable crash, bringing it within the scope of supervision. However, divergence detection is inherently best-effort—a process that performs useful work slowly is indistinguishable from one that is divergent—and the watchdog timeout must be tuned to the application's expected latency profile.

Resource exhaustion. System-level resource exhaustion (out of memory, file descriptor limits) can affect all processes simultaneously, potentially violating crash containment. Japl's runtime attempts to reserve emergency memory for crash handling, but this is a best-effort mechanism.

Error type proliferation. In large systems, the number of distinct error types can grow, leading to complex error type hierarchies. Japl mitigates this through error type composition (wrapping sub-errors in higher-level types) and the `map_err` pattern, but discipline is still required.

12.2 Relationship to Algebraic Effects Research

Japl's `Fail` effect is a specific instance of the general algebraic effects framework Plotkin and Pretnar [2009, 2013]. The effect handlers for `Fail` are relatively simple (the handler either returns the error or discards the continuation), which means that Japl does not require the full generality of effect handlers with resumable continuations.

This restriction is deliberate: the full algebraic effects model with multi-shot continuations introduces complexity (both conceptual and in implementation) that is not justified by the error handling use case. Japl uses one-shot continuations for error handling, which can be implemented without heap-allocated continuation frames.

Future work may extend Japl's effect system with more general algebraic effects for other use cases (generators, async/await, transactional memory), using the `Fail` effect as a proving ground.

12.3 The Category-Theoretic Perspective

The Kleisli category perspective provides more than just a formal framework; it suggests design principles. In $\mathbf{Kl}(T_E)$:

- Composition of fallible functions is automatically associative.
- Error propagation is a natural consequence of the monad structure.
- The relationship between pure functions (in **Set**) and fallible functions (in $\mathbf{Kl}(T_E)$) is captured by the adjunction between the free and forgetful functors.

The process algebra layer adds a second category: the category of processes and communication channels. The boundary between the two categories (domain errors and process failures) is not a functor—there is no structure-preserving map from process crashes to domain errors—which reinforces the design decision to keep them separate.

12.4 Educational Considerations

One of Japl's goals is accessibility. The dual error model is designed to be explainable without category theory:

1. **Domain errors**: "Sometimes things don't work out. Your function says so in its return type, and you have to deal with it." This is familiar to Rust and Go programmers.
2. **Process crashes**: "Sometimes things go really wrong. Your process dies, and a supervisor starts a fresh one." This is familiar to Erlang programmers and to anyone who has restarted a crashed application.

The effect system adds the compiler-checked documentation aspect: "The type signature tells you exactly what can go wrong and how." This is the lesson of Java's checked exceptions, done correctly.

13 Related Work: Extended Discussion

13.1 Algebraic Effects in Practice

Koka Leijen [2017] demonstrates that algebraic effects can be practical in a general-purpose language. Koka's effect system tracks all computational effects including exceptions, divergence, and IO. Japl's effect system is inspired by Koka's but differs in two important ways: (1) Japl adds process-level effects (`Process`), and (2) Japl's `Fail` effect is specialized for error handling rather than being a general exception effect.

The Eff programming language Bauer and Pretnar [2015] provides a direct implementation of algebraic effects and handlers with a clean operational semantics. Frank Lindley et al. [2017] explores effect typing with a focus on multi-handler patterns. Helium Biernacki et al. [2019] and Effekt Brachthäuser et al. [2020] explore efficient implementations of effect handlers.

13.2 Process Calculi and Typed Communication

The π-calculus Milner [1999] provides the foundation for reasoning about mobile processes and name-passing. Session types Honda [1993], Honda et al. [1998] extend this with typed communication protocols, ensuring that processes follow agreed-upon message exchange patterns. Japl's typed mailboxes can be viewed as a simplified form of session typing where the session protocol is determined by the mailbox type.

Singularity OS Hunt and Larus [2007] explored software-isolated processes with typed channels in a systems programming context, demonstrating that process isolation can be enforced by the type system rather than hardware memory protection.

13.3 Error Handling in Distributed Systems

The eight fallacies of distributed computing Deutsch [1994] highlight that network failures are fundamentally different from local failures. Waldo *et al.* Waldo et al. [1997] argue that distributed computing cannot be made transparent—the programmer must be aware of distribution boundaries.

Japl takes this critique seriously: the `Net` effect explicitly marks functions that involve network communication, and network failures are domain errors (represented as `Result` types) rather than crashes, since they are anticipated and recoverable. However, Japl allows process IDs to be location-transparent, accepting Waldo's critique for failure handling while maintaining uniformity for the communication model.

13.4 Gradual Typing and Error Handling

Gradual typing Siek and Taha [2006] offers a middle ground between static and dynamic typing. Similarly, one could imagine a "gradual error handling" approach where error types are optionally specified. Japl deliberately does not take this path: the value of the `Fail` effect comes precisely from its completeness. A "gradual" `Fail` effect that allows untracked errors would undermine the error safety theorem.

14 Future Work

Several directions for future research emerge from this work:

Automatic error type inference across module boundaries. Currently, Japl requires explicit error type annotations at module boundaries. Research into inferring error types across modules while maintaining modular compilation could reduce annotation burden.

Supervision strategy verification. Given typed crash reasons and supervision strategies, it may be possible to statically verify that a supervision tree handles all crash scenarios appropriately—that no crash reason goes unhandled and no restart strategy is inappropriately permissive.

Performance-guided error mode selection. Profiling data could guide the compiler in optimizing error handling paths—for example, converting `Result` checks to branch-free code when profiling shows the error path is extremely rare.

Cross-node supervision. Extending the supervision model to distributed settings, where supervisor and child may be on different nodes, introduces new challenges around failure detection timing and network partition handling.

Error type evolution. As systems evolve, error types change. Research into backward-compatible error type evolution (adding variants without breaking existing handlers) could leverage JAPL's row-polymorphic type system.

Integration with property-based testing. JAPL's built-in property testing could be extended to automatically generate test cases that exercise error paths, using the `Fail` effect annotations to guide test generation.

15 Conclusion

Error handling is not a peripheral concern—it is central to the design of reliable software. The history of programming languages shows that no single error handling mechanism is sufficient. Exceptions conflate control flow with failure signaling. Error codes lack algebraic structure. Result types handle domain errors well but provide no answer for unexpected failures. Crash-and-restart semantics handle unexpected failures well but lack compile-time error typing.

JAPL's dual error model synthesizes the best insights from each tradition:

- From **Rust**: typed result values for recoverable errors, the `?` propagation operator, exhaustive pattern matching on error types.
- From **Erlang**: process isolation, crash-and-restart semantics, supervision trees, the error kernel pattern.
- From **algebraic effects**: the `Fail` effect for compiler-tracked, composable failure handling.
- From **Java** (corrected): the insight that failure should be visible in type signatures, implemented through effects rather than checked exceptions.

The key contribution is the principled boundary between domain errors and process failures. Domain errors are expected, handled as values, tracked by the effect system, and resolved within the call chain. Process failures are unexpected, handled by supervision, and resolved by restarting with fresh state. The two modes are complementary, not competing.

We have shown that this model satisfies three important formal properties: error safety (domain errors cannot be silently discarded), crash containment (a process crash cannot corrupt another process), and supervision liveness (crashed permanent processes are eventually restarted). These properties hold compositionally, enabling modular reasoning about error handling in large systems.

Through case studies of an HTTP server, a database connection pool, and a distributed task worker, we have demonstrated that the dual model handles real-world error scenarios with minimal ceremony and maximal type safety. The result is a language that makes failure a first-class, well-typed, well-understood aspect of program design—because failures are normal, and they should be typed.

References

D. Abrahams. Exception safety in generic components. In *Generic Programming*, volume 1766 of *LNCS*, pages 69–79. Springer, 2000.

J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology, Stockholm, 2003.

J. Armstrong. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*, pages 6-1–6-26, 2007.

A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages*, 3(POPL):6:1–6:28, 2019.

J. Bloch. *Effective Java.* Addison-Wesley, 2nd edition, 2008.

J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):126:1–126:30, 2020.

T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

P. Deutsch. The eight fallacies of distributed computing. Technical report, Sun Microsystems, 1994.

A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015.

B. Eckel. Does Java need checked exceptions? `http://www.mindview.net/Etc/Discussions/CheckedExceptions`, 2003.

J. Epstein, A. P. Black, and S. Peyton Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 118–129, 2011.

J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley, 2014.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer, 1995.

S. King. polysemy: Higher-order, no-boilerplate, zero-cost free monads. Hackage library, 2019.

O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, pages 59–70, 2013.

S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2019.

D. 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.

T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 167–178, 2006.

S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 500–514, 2017.

M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning Publications, 2010.

S. Marlow, editor. *Haskell 2010 Language Report*. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 274–285, 2001.

N. D. Matsakis and F. S. Klock II. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, pages 103–104, 2014.

R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

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

R. Pike. Go at Google: Language design in the service of software engineering. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, 2012.

G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems (ESOP)*, volume 5502 of *LNCS*, pages 80–94. Springer, 2009.

G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4):1–36, 2013.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

H. Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.* Addison-Wesley, 1999.

P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–14, 1992.

P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.

J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 49–64. Springer, 1997.

N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. *Journal of Functional Programming*, 29:e16, 2019.

A Extended Typing Rules

We present the full typing rules for JAPL's error handling constructs.

A.1 Judgment Forms

The main typing judgment has the form:

$$\Gamma \vdash e : \tau \mid \mathcal{E}$$

where Γ is the type environment, e is an expression, τ is the result type, and $\mathcal{E}$ is the effect set.

A.2 Rules for Fail

Fail introduction.

$$\frac{\Gamma \vdash e : E}{\Gamma \vdash \mathsf{fail}(e) : \tau \mid \mathcal{E} \cup \{\texttt{Fail}[E]\}}$$

Fail propagation (? operator).

$$\frac{\Gamma \vdash e : \texttt{Result}\langle A, E\rangle \mid \mathcal{E}}{\Gamma \vdash e? : A \mid \mathcal{E} \cup \{\texttt{Fail}[E]\}}$$

Fail handling.

$$\frac{\Gamma \vdash e : A \mid \mathcal{E} \cup \{\texttt{Fail}[E]\}}{\Gamma \vdash \mathsf{Fail.catch}(\lambda().\, e) : \texttt{Result}\langle A, E\rangle \mid \mathcal{E}}$$

Effect subsumption.

$$\frac{\Gamma \vdash e : \tau \mid \mathcal{E}_1 \quad \mathcal{E}_1 \subseteq \mathcal{E}_2}{\Gamma \vdash e : \tau \mid \mathcal{E}_2}$$

A.3 Rules for Processes and Crashes

Process spawn.

$$\frac{\Gamma \vdash e : \mathsf{Never} \mid \mathcal{E} \cup \{\texttt{Process}[M]\}}{\Gamma \vdash \mathsf{Process.spawn}(\lambda().\, e) : \mathsf{Pid}[M] \mid \mathcal{E}' \cup \{\texttt{Process}\}}$$

Assert (crash on failure).

$$\frac{\Gamma \vdash e : \mathsf{Bool}}{\Gamma \vdash \mathsf{assert}\ e : \mathsf{Unit} \mid \mathcal{E}}$$

Note: `assert` does not add an effect—it either succeeds (producing `Unit`) or crashes the process. Process crashes are not tracked in the effect system because they are by definition unexpected.

Process monitor.

$$\frac{\Gamma \vdash p : \mathsf{Pid}[M]}{\Gamma \vdash \mathsf{Process.monitor}(p) : \mathsf{MonitorRef} \mid \mathcal{E} \cup \{\texttt{Process}\}}$$

A.4 Linear Resource Typing Rules

The following rules formalize the linear typing discipline that underpins deterministic resource cleanup and the Crash Containment proof (Theorem 11.2). A *linear type* $\tau^!$ denotes a resource that must be consumed (used or released) exactly once.

Resource acquisition.

$$\frac{\Gamma \vdash e_{args} : \tau_{args} \mid \mathcal{E} \cup \{\texttt{Io}\}}{\Gamma \vdash \mathsf{acquire}(e_{args}) : \tau^! \mid \mathcal{E} \cup \{\texttt{Io}\}}$$

The acquire operation produces a linearly-typed resource handle. The handle must appear in exactly one subsequent consumption site.

Resource release.

$$\frac{\Gamma, x : \tau^! \vdash e_{body} : \sigma \mid \mathcal{E}}{\Gamma \vdash \mathsf{release}(x, e_{body}) : \sigma \mid \mathcal{E}}$$

The release operation consumes the linear resource x, removing it from the environment. After release, x is no longer available.

Linear variable usage.

$$\frac{x : \tau^! \in \Gamma}{\Gamma \vdash x : \tau^! \mid \mathcal{E}} \quad \text{(exactly once in } \Gamma\text{)}$$

A linearly-typed binding must be used exactly once. The type checker rejects programs that use a linear variable zero times (resource leak) or more than once (use-after-free).

Ownership transfer (send).

$$\frac{\Gamma_1 \vdash p : \mathsf{Pid}[M] \quad \Gamma_2 \vdash v : \tau^! \quad \Gamma_1 \cap \Gamma_2 \text{ contains no linear bindings}}{\Gamma_1, \Gamma_2 \vdash \mathsf{Process.send}(p, v) : \mathsf{Unit} \mid \mathcal{E} \cup \{\mathtt{Process}\}}$$

Sending a linearly-typed value to another process transfers ownership: the sender's environment no longer contains the binding, and the receiver becomes the sole owner.

Crash cleanup obligation. When a process crashes, the runtime walks its stack and invokes the drop handler for every live linear binding. The type system guarantees that every $\tau^!$ in scope has a corresponding drop handler registered at allocation time. This is the formal basis for the "deterministic cleanup" claim in the Crash Containment proof: no resource is leaked, because every live linear binding is released by the unwinder.

A.5 Result Type Rules

Ok introduction.

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{Ok}(e) : \mathtt{Result}\langle A, E\rangle \mid \mathcal{E}}$$

Err introduction.

$$\frac{\Gamma \vdash e : E}{\Gamma \vdash \mathsf{Err}(e) : \mathtt{Result}\langle A, E\rangle \mid \mathcal{E}}$$

Match elimination.

$$\frac{\Gamma \vdash e : \mathtt{Result}\langle A, E\rangle \mid \mathcal{E} \quad \Gamma, x : A \vdash e_1 : \tau \mid \mathcal{E} \quad \Gamma, y : E \vdash e_2 : \tau \mid \mathcal{E}}{\Gamma \vdash \mathsf{match}\ e\ \{\mathsf{Ok}(x) \Rightarrow e_1;\ \mathsf{Err}(y) \Rightarrow e_2\} : \tau \mid \mathcal{E}}$$

B Supervision State Machine

A supervisor process can be modeled as a finite state machine with the following states:

1. **Running**: All children are alive and operational.
2. **Restarting**: A child has crashed and is being restarted.
3. **Escalating**: Restart limits have been exceeded; the supervisor is crashing.
4. **ShuttingDown**: The supervisor has received a shutdown signal and is terminating children.

The transitions are:

$$\mathsf{Running} \xrightarrow{\mathsf{ChildCrash}(i)} \mathsf{Restarting}(i)$$
$$\mathsf{Restarting}(i) \xrightarrow{\mathsf{WithinLimits}} \mathsf{Running}$$
$$\mathsf{Restarting}(i) \xrightarrow{\mathsf{LimitsExceeded}} \mathsf{Escalating}$$
$$\mathsf{Running} \xrightarrow{\mathsf{Shutdown}} \mathsf{ShuttingDown}$$
$$\mathsf{Escalating} \xrightarrow{\tau} \mathsf{crash}(\mathsf{MaxRestartsExceeded})$$

The restart limit check uses a sliding window:

```
fn check_restart_limit(
  history: List[Timestamp],
  max_restarts: Int,
  max_seconds: Int,
  now: Timestamp
) → Bool =
  let recent = List.filter(history,
    fn t →Timestamp.diff(now, t) < max_seconds)
  List.length(recent) < max_restarts
```

C Error Handling Patterns Catalog

We catalog common error handling patterns in Japl.

C.1 Pattern: Error Mapping

Convert between error types at module boundaries:

```
fn get_user(id: UserId) →User
    with Io, Fail[AppError] =
  let row = db_query(id)
    ▷ map_err(fn e →DbError(show(e)))?
  decode_user(row)
    ▷ map_err(fn e →InvalidInput(show(e)))?
```

C.2 Pattern: Error Accumulation

Collect multiple errors instead of failing at the first one:

```
fn validate_form(form: Form)
    → Result[ValidForm, List[FieldError]] =
  let errors = []
  let errors = match validate_name(form.name) with
    | Err(e) → [e, ..errors]
    | Ok(_) → errors
  let errors = match validate_email(form.email) with
    | Err(e) → [e, ..errors]
    | Ok(_) → errors
  let errors = match validate_age(form.age) with
    | Err(e) → [e, ..errors]
    | Ok(_) → errors
  if List.is_empty(errors) then
    Ok(to_valid_form(form))
  else
    Err(List.reverse(errors))
```

C.3 Pattern: Retry with Backoff

Retry a fallible operation with exponential backoff:

```
fn retry(
  max_attempts: Int,
  base_delay_ms: Int,
  action: fn() → a with Io, Fail[e]
) → a with Io, Fail[e] =
  fn go(attempt: Int) → a with Io, Fail[e] =
    match Fail.catch(action) with
    | Ok(result) → result
    | Err(e) →
        if attempt ≥ max_attempts then
          Fail.raise(e)
        else
          let delay = base_delay_ms
            * Int.pow(2, attempt)
          Process.sleep(delay)
          go(attempt + 1)
  go(0)
```

C.4 Pattern: Circuit Breaker

Prevent cascading failures by stopping calls to a failing service:

```
type CircuitState =
  | Closed(failure_count: Int)
  | Open(until: Timestamp)
  | HalfOpen

fn circuit_breaker(
  state: CircuitState,
  threshold: Int,
  timeout_ms: Int,
  action: fn() → a with Io, Net, Fail[e]
) → a with Io, Net, Fail[CircuitError[e]] =
  match state with
  | Open(until) →
      if Timestamp.now() < until then
        Fail.raise(CircuitOpen)
      else
        circuit_breaker(HalfOpen,
          threshold, timeout_ms, action)
  | HalfOpen →
      match Fail.catch(action) with
      | Ok(result) → result
      | Err(e) →
          Fail.raise(ServiceError(e))
  | Closed(count) →
      match Fail.catch(action) with
      | Ok(result) → result
      | Err(e) →
          if count + 1 ≥ threshold then
            Fail.raise(CircuitOpen)
          else
            Fail.raise(ServiceError(e))
```

C.5 Pattern: Crash Boundary

Convert between domain errors and process crashes at a well-defined boundary:

```
-- Wrap a potentially-crashing computation
-- as a domain error
fn safe_call(f: fn() → a with Io)
    → Result[a, String] with Process, Io =
  let worker = Process.spawn(fn →
    let result = f()
    Process.send(Process.self_parent(),
      WorkerDone(result))
  )
  Process.monitor(worker)
  match Process.receive_with_timeout(10000) with
  | WorkerDone(result) → Ok(result)
  | ProcessDown(^worker, reason) →
      Err("worker crashed: " ++ show(reason))
  | _ → Err("timeout waiting for worker")
```