Error Propagation in Pipeline-Native Languages

Typed short-circuit semantics, error accumulation, and recovery operators

Lateralus Language

bad-antics · January 2024 · Lateralus Language Research

ABSTRACT Error handling is the aspect of pipeline-native languages most frequently under-specified. Languages that treat the pipe operator as sugar provide no native mechanism for propagating errors through a pipeline; the programmer must manually thread Result monads or use exception-based control flow. Lateralus provides three error operators with distinct semantics: short-circuit propagation (|?>), error accumulation (|!>), and recovery (|~>). This paper specifies their typing rules, denotational semantics, and compilation strategy, and shows that together they cover all practical error-handling patterns without requiring a separate combinator library.

1. Why Error Handling Belongs in the Pipeline

In a traditional expression-oriented language, error handling interrupts the visual flow of data transformation. A five-step pipeline where each step can fail requires either nested conditionals, a chain of .andThen() calls, or exception-based control flow — all of which obscure the primary data flow.

Lateralus takes the position that error handling is not an interruption of the pipeline but a property of individual pipeline stages. The operator connecting two stages communicates how errors are handled at that boundary. The result is code where the primary path and the error paths are both visible at a glance.

1.1 Three Error Patterns

Practical code exhibits three distinct error patterns:

- **Short-circuit**: the first error stops the pipeline. Use case: request parsing, where there is no meaningful recovery from a malformed input.
- **Accumulation**: all errors are collected before short-circuiting. Use case: form validation, where the user wants to see all validation failures at once.
- **Recovery**: an error triggers a fallback computation. Use case: cache misses, optional enrichment steps, graceful degradation.

Each pattern maps to a distinct operator in Lateralus, keeping the code intent explicit without requiring a combinator library.

2. Short-Circuit Propagation: |?>

The |?> operator implements monadic bind over Result<T, E>. When the left-hand side evaluates to Ok(v), v is passed to the right-hand stage. When the left-hand side evaluates to Err(e), evaluation of the current pipeline terminates and the pipeline returns Err(e) immediately.

```
Gamma |- x : Result<A, E>    Gamma |- f : A -> Result<B, E>
-------------------------------------------------------------
              Gamma |- x |?> f : Result<B, E>
```

The compiler generates a conditional branch after each |?> stage: if the result tag is Err, jump to the pipeline's exit block with the error value. If the result tag is Ok, unwrap the value and pass it to the next stage. A sequence of N |?> stages generates N branch instructions, but a branch-elimination pass fuses consecutive error branches into a single early-exit block when the error types are identical.

```
// Five-stage request handler: all steps can fail
let response = request
    |?> parse_method
```

```
    |?> validate_auth_header
    |?> deserialize_body
    |?> apply_business_rules
    |?> serialize_response
```

After branch fusion, the compiler generates one error exit point, not five. The generated code is structurally equivalent to a hand-written if-chain but is produced automatically from the pipeline form.

3. Error Accumulation: |!>

The |!> operator accumulates errors from all stages before returning. The input type must be Result<A, E> and the stage function must return Result<B, E>. If the input is Err(e), the stage still executes with the wrapped value and any new error is appended to the error list. The final result is either Ok(v) if all stages succeeded or Err(errors) with the full list of failures.

```
Gamma |- x : Result<A, Vec<E>>    Gamma |- f : A -> Result<B, E>
-------------------------------------------------------------------
              Gamma |- x |!> f : Result<B, Vec<E>>
```

Note the asymmetry: the error type of the operator is Vec<E> (a list of errors), while individual stages return Result<B, E> (a single error). The accumulation operator lifts each stage's single error into the growing list.

```
// Form validation: collect all field errors before returning
let validated = form_data
    |!> validate_email
    |!> validate_phone
    |!> validate_address
    |!> validate_payment_method
// Returns Ok(form) or Err(["email invalid", "phone too short", ...])
```

3.1 Handling Dependent Fields

When later validation steps depend on earlier ones (e.g., the city field can only be validated if the country field is present), the programmer switches from |!> to |?> at the dependency boundary. The two operators are freely composable:

```
let result = form
    |!> validate_email
    |!> validate_phone
    |?> require_country    // must succeed before city check
    |!> validate_city
    |!> validate_postcode
```

4. Recovery Operator: |~>

The |~> operator implements error recovery: if the left side returns Err(e), the recovery function is called with e and its result is used as the pipeline value. If the left side returns Ok(v), the recovery function is not called.

```
Gamma |- x : Result<A, E>    Gamma |- f : E -> Result<A, F>
-------------------------------------------------------------
              Gamma |- x |~> f : Result<A, F>
```

The recovery function maps the old error type E to a new error type F, which allows error type transformations at recovery points.

```
// Cache-aside pattern: try cache, recover by querying DB
```

```
let data = cache_key
    |?> cache::lookup          // Err on miss
    |~> db::fetch_and_cache    // called only on miss

// Graceful degradation: enrich with ML model, recover with rule-based
let enriched = record
    |?> ml_enricher            // Err if model unavailable
    |~> rule_based_enricher    // fallback is always available
```

5. Operator Composition and Precedence

The three error operators compose freely with each other and with the total operator |>. The operator precedence is uniform (left-to-right evaluation in sequence order), which means the programmer controls error strategy at each stage boundary explicitly.

A pipeline can switch between accumulation and short-circuit within a single expression:

```
let result = input
    |?> parse            // short-circuit: no point accumulating parse errors
    |!> validate_field_a // accumulate: show all field errors
    |!> validate_field_b
    |!> validate_field_c
    |~> apply_defaults   // recover: fill in defaults if validation partially failed
    |>  serialize        // total: cannot fail
```

5.1 Error Type Unification

When mixing operators, the error types of consecutive stages must unify or be explicitly converted. The type checker reports a mismatch at the exact boundary where types diverge, not at the end of the pipeline, making error messages actionable.

6. Compilation Strategy

The compiler represents error-propagating pipelines as a CFG with an error corridor: a set of basic blocks connected exclusively by error-tagged branches. The happy path is a straight-line sequence of blocks; the error corridor collects all the off-ramps.

For |?> pipelines, the error corridor has a single merge point at the pipeline exit. For |!> pipelines, the error corridor carries a Vec<E> accumulator and each stage appends to it before continuing. For |~>, the error corridor branches into the recovery function rather than the exit.

```
// Compiler IR sketch for a |?> pipeline
bb0:
    %r0 = call parse(input)
    br %r0.is_ok, bb1, bb_exit_err
bb1:
    %r1 = call validate_auth(%r0.ok_val)
    br %r1.is_ok, bb2, bb_exit_err
bb2:
    %r2 = call deserialize_body(%r1.ok_val)
    br %r2.is_ok, bb3, bb_exit_err
bb_exit_err:
    // single shared error block — branch fusion result
    return Err(current_err)
```

7. Comparison with Existing Approaches

We compare against three existing error-handling styles: Haskell's do-notation, Rust's ? operator, and Java's checked exceptions. For each, we assess readability (is the data flow visible?), composability (can error strategies be mixed?), and performance (does the model impose runtime overhead?).

Style	Data Flow	Composable	Zero-Cost
Haskell do-notation	Medium	Yes	Yes
Rust ? operator	Low	No	Yes
Java checked exns	Low	No	No
Lateralus |?> |!> |~>	High	Yes	Yes

The 'Data Flow' column reflects the visual left-to-right readability of the transformation sequence. Lateralus scores highest because the pipeline form is preserved regardless of which error operators are used; Rust's ? clutters expressions and breaks the pipeline visual when stages return different Result types.

8. Conclusion

Error propagation in pipeline-native languages does not require a monadic library or a separate exception mechanism. Three typed operators — |?> for short-circuit, |!> for accumulation, and |~> for recovery — cover the full space of practical error patterns while remaining composable with each other and with the total pipeline operator |>.

The compilation strategy produces code equivalent to hand-written error handling with zero abstraction overhead. Future work: extending the accumulation operator to support partial results (returning both successful intermediate values and the accumulated error list) and integrating the recovery operator with the async pipeline for fault-tolerant streaming.