Higher-Order Pipelines in Lateralus

Pipelines as values, pipeline transformers, and combinator composition

Lateralus Language

bad-antics · December 2023 · Lateralus Language Research

ABSTRACT A pipeline in Lateralus is a first-class value. This means a pipeline can be passed as an argument, returned from a function, stored in a data structure, and composed with other pipelines. We call functions that accept or return pipelines 'pipeline transformers' and show that this abstraction subsumes several design patterns that require separate library abstractions in other languages: middleware chains, interceptor stacks, decorator pipelines, and retry/fallback wrappers. We give the typing rules for pipeline values, show how the compiler preserves optimization opportunities across higher-order boundaries, and benchmark three real-world use cases.

1. Pipelines as First-Class Values

In languages where |> is syntactic sugar, a 'pipeline' is not a value — it is a textual pattern in source code. You cannot write a function that accepts a pipeline and returns a modified version of it without resorting to continuation-passing style or reflection.

Lateralus pipelines are values of type Pipeline<A, B>, where A is the input type and B is the output type. A pipeline literal is written with the pipe { } keyword, and a pipeline expression can be bound to a name like any other value:

```
// A pipeline value — can be stored, passed, composed
let validate_and_enrich : Pipeline<RawRequest, EnrichedRequest> = pipe {
    |?> parse_headers
    |?> validate_auth
    |>  enrich_with_session
    |>  attach_trace_id
}
```

The pipeline value is separate from its invocation. To run a pipeline, you use the application syntax input |> pipeline or the explicit call pipeline.run(input). This separation between definition and execution is what enables higher-order composition.

2. Pipeline Transformers

A pipeline transformer is a function with signature (Pipeline<A, B>) -> Pipeline<A, B> or (Pipeline<A, B>) -> Pipeline<C, D>. Transformers modify a pipeline's behavior without access to its internal stages. This models the 'middleware' or 'interceptor' pattern that web frameworks implement with mutable stacks.

2.1 Logging Transformer

A logging transformer wraps each stage of a pipeline with entry/exit logging without modifying the stages themselves:

```
fn with_logging(p: Pipeline<A, B>) -> Pipeline<A, B> {
    pipe {
        |> log::enter
        |> p          // the original pipeline as a stage
        |> log::exit
    }
}

// Usage
let logged_pipeline = validate_and_enrich |> with_logging
let result = request |> logged_pipeline
```

2.2 Retry Transformer

A retry transformer wraps a pipeline with exponential backoff on failure. Because |?> propagates errors, the retry logic only activates when the inner pipeline returns Err:

```
fn with_retry(max: u32, p: Pipeline<A, Result<B, E>>) -> Pipeline<A, Result<B, E>> {
    pipe {
        |?> p
        |?> retry::on_err(max, backoff::exponential(100ms))
    }
}

let resilient = http_fetch_pipeline |> with_retry(3)
```

3. Typing Rules for Pipeline Values

A pipeline value p : Pipeline<A, B> is typed as a function from A to B with the additional constraint that the body is a sequence of pipeline-operator expressions. The type checker assigns a kind Pipeline to distinguish it from bare function types; this prevents a pipeline value from being called without the run context that handles backpressure and async scheduling.

```
stage_1 : A -> R1    stage_2 : R1 -> R2    ...    stage_n : R(n-1) -> B
-------------------------------------------------------------------------
 pipe { |> stage_1 |> stage_2 ... |> stage_n } : Pipeline<A, B>
```

Composition of two compatible pipelines uses the >> pipeline composition operator:

```
p : Pipeline<A, B>    q : Pipeline<B, C>
-----------------------------------------
         p >> q : Pipeline<A, C>
```

This rule mirrors function composition but preserves the pipeline kind, so the composed value retains its optimizer metadata (stage list, variant sequence, fusion hints).

4. Composition Operators

Beyond sequential composition (>>), Lateralus provides three additional composition operators for pipeline values:

- **||** — parallel composition: p || q : Pipeline<(A, C), (B, D)> runs two independent pipelines on independent inputs simultaneously.
- **&&** — fan-in: p && q : Pipeline<A, (B, C)> sends the same input to two pipelines and collects both outputs.
- **??** — fallback: p ?? q : Pipeline<A, Result<B, E>> tries p and falls back to q on failure.

```
// Fan-in: validate with two independent validators, get both results
let dual_validate = schema_validator && auth_validator
// result type: Pipeline<Request, (SchemaResult, AuthResult)>

// Fallback: try fast cache, fall back to slow DB
let fetch = cache_fetch ?? db_fetch
// On cache miss (Err), automatically tries db_fetch
```

The fallback operator is particularly useful in combination with the retry transformer: (primary ?? fallback) |> with_retry(3) retries the entire primary-then-fallback sequence up to three times before giving up.

5. Middleware Stacks Without Mutation

Web frameworks conventionally implement middleware as a mutable stack: each middleware function pushes itself onto a list, and the framework iterates the list at request time. This couples middleware ordering to mutation and makes the final pipeline shape invisible at compile time.

In Lateralus, a middleware stack is a pipeline value built by sequential composition. The stack is immutable; adding a middleware produces a new pipeline value rather than mutating the existing one:

```
let base = pipe { |> router::dispatch }

let with_auth    = auth_middleware    >> base
let with_logging = logging_middleware >> with_auth
let with_cors    = cors_middleware    >> with_logging

// The complete pipeline is a value; its shape is known at compile time
server::serve(with_cors)
```

Because the full pipeline is constructed before any request arrives, the compiler can fuse the middleware stages that are marked fusable and generate a single function for the common request path. At runtime, the 'middleware ' overhead is zero for hot requests that hit the fast path.

6. Decorator Pattern Without Reflection

Object-oriented languages implement the Decorator pattern with inheritance or wrapping, which requires runtime dispatch and prevents inlining. Lateralus implements decoration as a pipeline transformer: a function that takes a pipeline, wraps it in pre/post stages, and returns the wrapped pipeline.

```
fn timed<A, B>(label: str, p: Pipeline<A, B>) -> Pipeline<A, B> {
    let start = metric::Timer::new(label)
    pipe {
        |> start.begin
        |> p
        |> start.end
    }
}

fn cached<A, B: Hash + Eq>(ttl: Duration, p: Pipeline<A, B>) -> Pipeline<A, B> {
    let cache = Cache::new(ttl)
    pipe { |?> cache.get_or_run(p) }
}

let hot_path = db_query
    |> timed("db_query")
    |> cached(30s)
```

The timed and cached transformers compose: p |> timed("x") |> cached(30s) produces a cached, timed pipeline. No inheritance hierarchy, no interface, no reflection.

7. Optimization Across Higher-Order Boundaries

The main risk of higher-order pipelines is that the compiler loses optimization information when a pipeline is passed through a function boundary. Lateralus addresses this through two mechanisms: specialization and pipeline inlining.

7.1 Specialization

When a pipeline transformer is called with a pipeline literal (not a variable), the compiler specializes the transformer for that literal. The specialization copies the transformer body and substitutes the concrete pipeline, exposing all stage metadata for fusion analysis.

7.2 Pipeline Inlining

When a pipeline is passed to a function that calls p.run(x) on it, the compiler inlines the pipeline body at the call site, equivalent to inlining a function closure. This means the abstract 'apply this pipeline' becomes a concrete stage sequence in the generated IR.

```
// After inlining and fusion, the transformer overhead disappears:
// with_logging(validate_and_enrich) becomes approximately:
pipe {
    |> log::enter
    |?> parse_headers
    |?> validate_auth
    |>  enrich_with_session
    |>  attach_trace_id
    |> log::exit
}  // single fused stage set
```

8. Benchmarks

We measured three use cases: a six-middleware HTTP handler, a four-stage compiler pass with decoration, and a ten-stage data pipeline with caching and retry. We compared first-class pipeline composition against equivalent code using a mutable middleware list (simulated in C for fairness).

Workload	Lateralus	Mutable Stack	Speedup
HTTP handler (6 middleware)	1.8 µs/req	4.2 µs/req	2.3×
Compiler pass (4 stage)	0.9 µs	2.1 µs	2.3×
Data pipeline (10 stage)	3.1 µs	8.7 µs	2.8×

The speedup comes primarily from fusion: the Lateralus compiler merged consecutive fusable stages into single generated functions, eliminating intermediate allocations and dispatch overhead. The mutable stack baseline was hand-optimized C; the comparison is conservative.