Pipelines Are Not Sugar

Why the |> operator demands a first-class semantic model

Lateralus Language

bad-antics · November 2023 · Lateralus Language Research

ABSTRACT Every mainstream language that has adopted a pipe operator treats it as syntactic sugar over nested function application. We argue this framing is not merely insufficient but actively harmful: it prevents meaningful optimizations, conflates error-propagating and non-error-propagating composition, and makes asynchronous pipelines an afterthought bolted onto a synchronous model. Lateralus instead treats the pipeline operator as a primitive semantic form with its own typing rules, control-flow semantics, and compiler IR nodes. This paper explains the distinction, gives formal justification, and shows empirically that a first-class model enables optimizations that sugar-based approaches cannot express.

1. The Sugar Framing and Its Costs

In F#, Elixir, and the TC39 JavaScript proposal, x |> f desugars to f(x) before any type-checking or optimization occurs. The pipeline operator is invisible to the type system: it cannot carry its own type variables, it cannot distinguish a function that returns Result from one that does not, and it cannot express the boundary between synchronous and asynchronous execution.

This has three measurable consequences. First, error propagation requires a separate combinator (Result.bind, Option.andThen), breaking visual flow whenever a step can fail. Second, async pipelines must manually thread await at each stage, making the structure of a streaming computation invisible to the compiler's fusion pass. Third, because desugaring happens before optimization, the compiler has no IR node representing 'this sequence of steps forms a pipeline' and cannot apply pipeline-specific transforms such as stage fusion, backpressure insertion, or dead-stage elimination.

1.1 Demonstrating the Desugaring Limit

Consider a four-stage transformation: parse → validate → enrich → serialize. In a sugar model, each stage is a separate call expression. The compiler sees four independent function applications and must rely on inlining and escape analysis to discover that intermediate values flow only downward. In a first-class model, the compiler sees one pipeline node with four stage slots; fusion is a single IR rewrite that eliminates all intermediate allocations without needing to inline across function boundaries.

```
// Sugar model: four independent calls (compiler sees no pipeline shape)
let result = serialize(enrich(validate(parse(input))))

// Lateralus first-class model: one IR node, four named stages
let result = input
    |>  parse         // can fail: desugar is WRONG here
    |?> validate      // |?> = propagate Err, continue on Ok
    |>  enrich
    |>> serialize     // |>> = async stage, returns Future<T>
```

2. The Four Variants and Their Semantics

Lateralus distinguishes four pipeline operator variants, each with its own denotational semantics:

- **|>** — total pipeline: x |> f requires f : A -> B. No failure, no async.
- **|?>** — error-propagating pipeline: x |?> f requires x : Result<A, E> and f : A -> Result<B, E>. On Err, evaluation short-circuits.
- **|>>** — async pipeline: x |>> f where f : A -> Future<B>. Chains futures without explicit await.
- **|>|** — fan-out pipeline: x |>| [f, g, h] forks one input to multiple stages running in parallel.

The key property is that each variant is a distinct IR node. The type checker assigns different constraints to each; the optimizer applies different rewrite rules; the code generator emits different runtime primitives. No amount of macro expansion or desugaring can achieve this without replicating the semantic machinery inside the macro.

2.1 Typing Rules

The typing judgment for |?> is:

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

This cannot be expressed as a polymorphic binary operator without adding higher-kinded type variables to the language, which introduces significant complexity for a primitive that appears on every line of practical code. Making it a syntactic keyword keeps the type rule simple and the error messages precise.

3. Compiler IR Representation

Lateralus IR represents a pipeline as a PipelineNode with an ordered list of StageDescriptor entries. Each descriptor records the stage function reference, the variant (total/error/async/fanout), the inferred input and output types, and a set of optimization hints (fusable, pure, side-effecting).

```
// Internal compiler IR (abbreviated)
PipelineNode {
    stages: [
        StageDescriptor { fn: parse,    variant: Total,   fusable: true  },
        StageDescriptor { fn: validate, variant: Error,   fusable: true  },
        StageDescriptor { fn: enrich,   variant: Total,   fusable: false },
        StageDescriptor { fn: serialize, variant: Async,  fusable: true  },
    ],
    input_type:  RawBytes,
    output_type: Future<SerializedJson>,
}
```

The fusion pass walks consecutive fusable stages of the same variant and merges them into a single generated function. The async scheduler pass identifies stage boundaries where a |>> appears and inserts yield points and continuation captures. Neither pass is possible without the explicit IR representation.

3.1 Dead-Stage Elimination

Because the pipeline shape is explicit, the optimizer can apply dead-stage elimination: if an output type annotation downstream of stage N is inconsistent with the output of stage N-1, the stage is flagged as unreachable before runtime. This catches whole classes of logic errors that sugar-based models defer to runtime type errors or silent data corruption.

4. Error Propagation Without Noise

The most common objection to Rust-style Result is that it clutters call sites with ? operators or .unwrap() calls. In Lateralus, |?> is the callsite annotation: the programmer writes the pipeline and the compiler inserts the propagation logic automatically.

Compare the equivalent code in three languages for a four-step pipeline where every step can fail:

```
-- Haskell (do-notation): correct but not visually a pipeline
```

```
result = do
    a <- parse input
    b <- validate a
    c <- enrich b
    serialize c

// Rust (? operator): correct, minimal noise
let result = serialize(enrich(validate(parse(input)?)?)?);

// Lateralus (|?> operator): correct, pipeline-visual
let result = input |?> parse |?> validate |?> enrich |?> serialize
```

The Lateralus form is visually left-to-right, requires no nesting, and does not scatter ? inside expressions. More importantly, the compiler knows the entire sequence is a single error-propagating pipeline and can generate a single error path rather than N separate branch targets.

4.1 Error Type Propagation

If stages return different error types, the compiler requires an explicit conversion. This is surfaced as a type error at the pipeline boundary rather than buried inside a combinator chain, making the source of incompatibility visible at the point of authorship.

5. Async Without Await Noise

Async/await syntax was introduced in most languages to flatten callback pyramids. In a sugar model, each async call in a pipeline still requires an await annotation, meaning the programmer must track which stages are async and annotate each one. In Lateralus, |>> marks the stage as async and the continuation is handled by the compiler.

```
// JavaScript async pipeline (await at every step)
const result = await serialize(await enrich(await validate(await parse(input))));

// Lateralus: async is a stage property, not a call-site annotation
let result = input |> parse |?> validate |>> enrich |>> serialize
```

The compiler generates a state machine that yields between |>> stages. The programmer sees a linear pipeline; the runtime sees a resumable coroutine. Dead-stage elimination applies to async stages as well: if an async stage produces a value that no subsequent stage consumes, the await is elided entirely.

6. Fan-Out Semantics

The |>| operator distributes one value to multiple independent stages, collecting results into a tuple or a product type. This pattern is common in validation (run N validators in parallel, collect all failures) and in multi-format serialization (emit JSON and Protobuf from the same in-memory value).

```
let diagnostics = ast
    |>| [
        lint::unused_variables,
        lint::shadow_warnings,
        lint::type_annotation_coverage,
    ]   // result: (DiagList, DiagList, DiagList)
```

Sugar cannot express fan-out without a combinator library. A combinator approach requires the programmer to construct a tuple of functions, pass the input to each, and destructure the output — four operations that the |>| keyword performs implicitly and that the compiler can parallelize at the thread-pool level when stages are pure.

7. Empirical Comparison

We compared code size and performance across three workloads: a five-stage JSON-to-Protobuf converter, a four-stage HTTP request validator, and a six-stage compiler pass. In each case we implemented the workload in Lateralus (using first-class pipeline operators) and in an equivalent Elixir pipeline (syntactic sugar). We measured source line count, binary size, and throughput.

```
Workload                  Lang       SLOC  Throughput (K req/s)
-----------------------------------------------------------------
JSON-to-Protobuf (5 stage) Lateralus    41        890
JSON-to-Protobuf (5 stage) Elixir        63        340
HTTP validator  (4 stage)  Lateralus    29        1240
HTTP validator  (4 stage)  Elixir        48        510
Compiler pass   (6 stage)  Lateralus    57        620
Compiler pass   (6 stage)  Elixir        92        210
```

The Lateralus numbers reflect stage-fusion optimization. The Elixir numbers use idiomatic |> with with blocks for error propagation. The throughput difference in the async workloads is primarily attributable to the compiler-generated coroutine state machine vs. the Elixir VM scheduler overhead, not to the pipeline syntax itself.

7.1 Threats to Validity

Both implementations were written by the same author, so there is a risk of unconscious bias toward the Lateralus form. We have open-sourced both implementations and invite independent replication. The benchmark harness uses process isolation with a 30-second warmup per configuration.

8. Related Work and Conclusion

Point-free style in Haskell achieves some of the compositional benefits of pipeline operators but at the cost of readability for non-Haskell programmers and with no native async model. Kotlin's coroutine flow API provides a pipeline-like abstraction but requires importing a library and offers no compiler-level fusion. Rust's iterator adapters are fusion-optimized but apply only to pull-based sequences, not to general function composition.

Lateralus occupies a distinct point: a minimal surface (four operators) with a first-class semantic model. The operators are not syntactic conveniences; they are typed forms that the compiler understands at every level from type checking through code generation. The result is more expressive than sugar, more readable than combinator libraries, and more optimizable than either.

Future work: extending the fan-out operator to model backpressure-aware streaming pipelines, and formalizing the interaction between the |>>| (async fan-out) operator and structured concurrency primitives.

Lateralus is an open-source, zero-dependency programming language. Project home: https://lateralus.dev. Source: github.com/bad-antics/lateralus-lang.