Pipeline Semantics: An Algebraic Treatment

Equational laws, rewrite systems, and algebraic optimization of Lateralus pipelines

Lateralus Language

bad-antics · June 2024 · Lateralus Language Research

ABSTRACT We present an equational theory for Lateralus pipeline expressions. The theory consists of 14 laws governing the four pipeline operators and their interactions. We show that the laws form a convergent term-rewriting system: every pipeline expression has a unique normal form, and the rewriting system reaches that normal form in polynomial time in the number of stages. The optimizer's fusion pass, dead-stage elimination, and operator hoisting are all instances of rewriting to normal form. We prove the rewriting system is confluent and terminating.

1. An Equational Theory for Pipelines

An equational theory is a set of equations between program expressions that the compiler is permitted to treat as equivalent. A well-designed equational theory has two properties: soundness (the equations are semantically valid) and utility (they correspond to useful optimizations).

For Lateralus pipelines, the 14 laws fall into four groups: associativity/identity laws, commutativity laws for concurrent stages, absorption laws for error operators, and distributivity laws for fan-out over error operators.

2. Associativity and Identity Laws

The total pipeline operator is associative (sequential execution order does not matter for termination) and has a two-sided identity:

```
-- Law 1: Associativity of |>
(p |> q) |> r = p |> (q |> r)

-- Law 2: Left identity
identity |> p = p

-- Law 3: Right identity
p |> identity = p
```

The same laws hold for |?> with the identity being Ok-wrapping:

```
-- Law 4: Associativity of |?>
(p |?> q) |?> r = p |?> (q |?> r)

-- Law 5: Left identity for |?>
ok_wrap |?> p = p     where ok_wrap x = Ok(x)

-- Law 6: Right identity for |?>
p |?> ok_wrap = p
```

Laws 4-6 establish that |?> forms a monoid over Result-returning stages, which is the algebraic counterpart of the categorical Kleisli composition.

3. Absorption Laws for Error Operators

The error propagation operator absorbs errors: once a stage returns Err, all subsequent |?> stages are skipped:

```
-- Law 7: Error absorption
Err(e) |?> f = Err(e)    for any stage f
```

```
-- Law 8: Ok threading
Ok(v) |?> f = f(v)
```

Law 7 is the foundation of the compiler's early-exit optimization: a sequence of |?> stages can be compiled as a single conditional chain with one exit block, because all stages after the first error are provably unreachable with that error value.

3.1 Recovery Absorption

The recovery operator is dual to error absorption:

```
-- Law 9: Ok pass-through for |~>
Ok(v) |~> f = Ok(v)     (recovery not invoked)

-- Law 10: Error recovery
Err(e) |~> f = f(e)
```

Laws 9 and 10 allow the optimizer to hoist recovery stages: if the preceding computation always succeeds, the recovery stage is dead code and can be eliminated.

4. Distributivity: Fan-Out over Error

The fan-out operator distributes over the error operators under certain conditions:

```
-- Law 11: Fan-out distributes over total |>
x |>| [f, g] |> h = x |>| [f |> h, g |> h]
  (if h is pure and does not depend on the fan-out pairing)

-- Law 12: Fan-out preserves |?> errors
x |>| [f, g] |?> h = (x |>| [f, g]) |?> h
  (error from fan-out propagates before h is called)
```

Law 11 enables stage hoisting: a postfix total stage that applies independently to each fan-out result can be pushed inside the fan-out, enabling parallel execution of the combined stage.

5. Commutativity of Concurrent Stages

Two stages are concurrent if they have no data dependency and produce no observable side effects relative to each other. Concurrent stages can be reordered:

```
-- Law 13: Commutativity of pure concurrent stages
If pure(f) and pure(g) and independent(f, g):
    x |> f |> g = x |> g |> f
    (any interleaving is observationally equivalent)
```

The compiler determines independence by data-flow analysis: if stage g does not read any value written by stage f other than the pipeline-passed value, they are independent. This law enables the scheduler to reorder stages for cache locality or to balance thread load.

6. The Rewriting System

The 14 laws define a term-rewriting system where each law is a left-to-right rewrite rule applied from the innermost subexpression outward. We prove the system is:

- **Terminating**: each rewrite strictly reduces a complexity measure (the number of Err-valued subexpressions in non-absorbed positions).

- **Confluent**: if two rewrites can both apply to the same expression, the resulting expressions can both be reduced to the same normal form (by the Church-Rosser property).

Together, termination and confluence guarantee that every expression has a unique normal form and that the optimizer reaches it regardless of the order in which it applies rewrite rules.

6.1 Normal Form Structure

A pipeline expression in normal form has the structure: a sequence of total stages (if any), followed by an optional error operator block (if any stages can fail), followed by a sequence of total stages (post-error). This matches the CFG structure the compiler generates: happy path, error corridor, post-recovery path.

7. Optimizer Correspondence

Each optimization pass in the Lateralus compiler corresponds to an application of one or more rewriting laws:

```
Optimization pass          Law(s) applied
-------------------------------------------------
Stage fusion               Laws 1, 4 (associativity)
Dead-stage elimination     Laws 7, 9 (absorption)
Fan-out hoisting           Law 11 (distributivity)
Error exit merging         Law 7 (absorption, repeatedly)
Concurrent reordering      Law 13 (commutativity)
Identity elimination       Laws 2, 3, 5, 6
```

Because each pass is an instance of law application, the passes are individually sound (each law is semantically valid) and the composition of passes is sound (confluence guarantees the same normal form regardless of order).

8. Law 14: The Pipeline Extraction Law

The final law is the most powerful: it allows the compiler to extract a pipeline value from a higher-order context without evaluation:

```
-- Law 14: Pipeline extraction
transformer(pipe { |> f |> g }) = pipe { |> f |> g |> transformer_suffix }
  (when transformer is a pipeline transformer of the form:
   fn transformer(p) { pipe { |> p |> suffix } })
```

This law enables the optimizer to see through pipeline transformer calls and expose the inner stages to fusion analysis. Without Law 14, a higher-order pipeline would be an opaque value that blocks all cross-stage optimizations. With it, the transformer is inlined and the combined stage list is available for all other rewrites.

9. Conclusion

The 14-law equational theory gives the Lateralus optimizer a rigorous foundation: every transformation it applies is an instance of a proven-sound law, and the confluence proof guarantees that multiple passes applied in any order produce the same result. This makes the optimizer correct by construction for the class of transformations covered by the theory.

Future work: extending the theory to cover the async operator |>>, which requires reasoning about concurrent execution and scheduler semantics beyond the purely functional model.