Type System v1.5: Extended Inference

Effect types, lifetime inference, and pipeline-aware type propagation

Lateralus Language

bad-antics · November 2025 · Lateralus Language Research

ABSTRACT Lateralus v1.5 extends the type system with three features: effect types that track I/O and mutation side effects, lifetime inference for borrow checking without explicit annotations, and pipeline-aware type propagation that infers the output type of a pipeline from its stage sequence. This paper specifies the new type rules, describes the inference algorithm extensions, and evaluates the impact on programmer-visible type annotations required in typical Lateralus code.

1. Effect Types

An effect type tracks what side effects a function or pipeline may perform. In Lateralus v1.5, effects are part of the function type:

```
// Function type with effects
fn read_file(path: &Path) -> Vec<u8> !{io}        // performs I/O
fn sort_in_place(xs: &mut Vec<i32>) !{mut}       // mutates
fn add(x: i32, y: i32) -> i32                     // pure, no effects
```

The effect set is part of the type; a function expecting a pure stage cannot accept an I/O-performing one. This enables the compiler to distinguish pure from effectful pipeline stages and apply pure-stage optimizations (CSE, fusion, reordering) only to stages without effects.

2. Effect Inference

Effect types are inferred automatically in most cases. The inference rule is: a function's effect set is the union of the effect sets of the expressions in its body.

```
  -- Effect inference
  Gamma |- e1 : T1 !{eff1}    Gamma |- e2 : T2 !{eff2}
  ------------------------------------------------------
        Gamma |- (e1; e2) : T2 !{eff1 ∪ eff2}
```

For pipeline stages, the effect set propagates: a pipeline with any effectful stage has the union of all stages' effect sets. This allows the scheduler to decide whether stages can be run in parallel (only if their effect sets are disjoint).

```
// Pipeline effect inference
let p = input
    |>  parse_pure        // !{}
    |>  log_to_file       // !{io}
    |>  transform_pure    // !{}
// Inferred effect: p has type Pipeline<A, B> !{io}
```

3. Lifetime Inference

Lateralus v1.5 introduces lifetime inference: the compiler infers borrow lifetimes without requiring the programmer to write lifetime annotations in most cases. Lifetime annotations are still accepted for cases where inference is ambiguous.

```
// v1.0: explicit lifetime annotations required
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

// v1.5: lifetime inferred from usage
fn longest(x: &str, y: &str) -> &str {
```

```
    if x.len() > y.len() { x } else { y }
}
```

The lifetime inference algorithm is based on the constraint generation approach of Polonius: for each borrow, generate a constraint that the borrow must not outlive the borrowed value, and solve the constraints using a fixed-point iteration.

4. Lifetime Inference for Pipelines

Pipeline values may capture borrows from their enclosing scope. The lifetime of a pipeline value must not exceed the lifetime of the shortest-lived borrow it captures:

```
let data = load_data();
let pipeline = pipe {
    |> filter_with(&data)  // borrows 'data'
    |> process
};
// 'pipeline' lifetime is bounded by 'data's lifetime
// Compiler error if pipeline escapes 'data's scope
```

The inference correctly handles this case: the pipeline type is annotated with a lifetime bound derived from the captured borrow, and the type checker verifies that the pipeline does not escape the scope.

5. Pipeline-Aware Type Propagation

Type propagation in v1.5 is bidirectional for pipeline expressions: the expected type at the end of a pipeline (from a type annotation or a consuming expression) propagates backward through the stage sequence, refining type inference for each stage.

```
// Backward type propagation
let result: Vec<ProcessedRecord> = input
    |> map(transform)    // transform is inferred as: ? -> ProcessedRecord
    |> collect()         // collect is inferred as: Iter<ProcessedRecord> -> Vec
```

Without backward propagation, the type of transform must be fully specified before the collection type is known. With backward propagation, the annotation on result refines the inferred types of all pipeline stages.

6. Gradual Effect Types

Gradual effect types allow dyn values to have an unknown effect set. A function that accepts a dyn callback must assume it has the maximal effect set !{io, mut, panic}.

This interacts with the optimization machinery: a pipeline stage that calls a dyn function cannot be fused with adjacent pure stages because its effects are unknown at compile time. The programmer can opt into fusion by providing a concrete function value with a known effect set.

7. Annotation Requirements: v1.0 vs v1.5

We measured the number of explicit type annotations required in a 5,000-line Lateralus codebase under v1.0 and v1.5 inference.

```
Annotation type          v1.0    v1.5   Reduction
-----------------------------------------------------
Lifetime annotations      127       8        94%
Effect annotations          0      12       N/A (new)
```

```
Return type annotations    89      31        65%
Generic type arguments    203      67        67%
Total annotations         419     118        72%
```

The 72% reduction in total annotations is primarily from lifetime inference. Effect annotations are new in v1.5 and represent a net increase in annotation burden for effectful functions, but they provide new safety guarantees in exchange.

8. Future Type System Work

Planned extensions for Lateralus v2.0:

- **Linear types**: a value that must be used exactly once, modeled as a type-level assertion. Useful for capabilities, file handles, and communication channels.
- **Dependent types (simple subset)**: types that depend on values, enabling array bounds checking at compile time and proving properties about pipeline output sizes.
- **Effect handlers**: first-class handlers that intercept effects (similar to algebraic effect handlers in Koka/Eff), enabling testable I/O.

The type system is designed for incremental extension: v1.5's effect types and lifetime inference are backward-compatible additions that do not change the semantics of v1.0 programs.