Type Inference in Lateralus 1.5

Algorithm W with bidirectional refinements and pipeline-aware unification

Lateralus Language

bad-antics · April 2026 · Lateralus Compiler Internals

ABSTRACT Lateralus 1.5 ships a gradual type system built on a variant of Algorithm W with bidirectional refinements, first-class pipeline types, and principal inference for generic functions. This paper walks through the implementation as it sits in lateralus_lang/type_inference.py, covering the core unifier, let-generalization, the occurs check, the row-polymorphic treatment of records, and the additions that make pipeline expressions composeable without explicit annotations. We close with the test strategy that gave us confidence to ship inference on by default and with the escape hatches we kept for programs that fall outside the inferable fragment.

1. Design Goals

Lateralus positions itself as a gradual language: fully-typed code should feel as precise as ML, while lightly-typed pipelines should require nothing beyond what the programmer would naturally write. The design goals for the 1.5 inference pass were:

- **No annotations on local bindings.** let x = expr should always infer.
- **Let-generalization.** let id = fn(x) { x } should be usable at multiple types in the same scope.
- **Principal types.** Every inferable program must have a unique most-general type; implementation must return it.
- **Pipeline composition.** xs |> map(f) |> filter(p) must flow types through |> without manual help.
- **Graceful fallback.** When inference reaches the edge of the decidable fragment (recursive records, higher-rank types), the compiler should emit a specific diagnostic rather than a silent any.

2. Core Types

The type language is small:

```
type Ty =
    | TyVar id:int                          # unification variable
    | TyCon name:str                        # int, str, float, bool, unit
    | TyApp head:str args:list[Ty]          # list[int], map[str,int]
    | TyFn params:list[Ty] ret:Ty           # (int, int) -> int
    | TyRecord fields:map[str,Ty] row:RowVar # {x:int, y:int | r}
    | TyForall vars:list[int] body:Ty       # only at let-binding
```

Unification operates on TyVar, TyCon, TyApp, TyFn, and TyRecord. TyForall appears only in type schemes stored in the environment and is instantiated on lookup.

3. The Unifier

The unifier follows the textbook shape:

```
fn unify(s: Subst, a: Ty, b: Ty) -> Subst:
    a = apply(s, a); b = apply(s, b)
    match (a, b):
        TyVar(i), TyVar(j) if i == j => s
        TyVar(i), t                  => bind(s, i, t)
        t, TyVar(i)                  => bind(s, i, t)
        TyCon(n), TyCon(m) if n == m => s
        TyApp(h1, a1), TyApp(h2, a2) => unify_list(s, h1==h2, a1, a2)
        TyFn(p1, r1), TyFn(p2, r2)   => unify_list(unify(s, r1, r2), p1, p2)
```

```
        TyRecord(f1,r1), TyRecord(f2,r2) => unify_record(s, f1, r1, f2, r2)
        _ => fail "cannot unify {a} with {b}"
```

The bind helper performs the occurs check: if i occurs in t: fail. Without this check, self-referential constraints like a ~ list[a] would produce an infinite substitution. In 1.5 we moved the check into the substitution-application path; it runs at bind time rather than at every application, which cut inference time on large pipeline expressions by roughly 30%.

4. Let-Generalization

On let name = expr in body, the inferred type of expr under substitution s is generalized by abstracting free variables that do not appear free in the substitution-applied environment:

```
fn generalize(env: Env, s: Subst, t: Ty) -> Scheme:
    let env_free = free_vars(apply(s, env))
    let quantified = free_vars(apply(s, t)) - env_free
    return TyForall(quantified, apply(s, t))
```

On lookup, a scheme is instantiated with fresh type variables: TyForall([a, b], a -> b) becomes c -> d for fresh c, d. This is what lets let id = fn(x) {x}; id(1); id("hi") type-check without annotation.

Crucially, generalization only happens at let. Lambda parameters are monomorphic within their body, preserving decidability.

5. Pipeline Inference

The pipeline operator |> is left-associative and has lower precedence than function application. Lateralus treats x |> f as shorthand for f(x) when f is a reference, and as shorthand for fn_of(f)(x) when f is a partial application like map(g). Inference must disambiguate without running the program:

```
fn infer_pipe(env, lhs, rhs):
    let t_lhs = infer(env, lhs)
    match rhs:
        Ident(f):
            let t_f = instantiate(lookup(env, f))
            let t_ret = fresh_ty()
            unify(t_f, TyFn([t_lhs], t_ret))
            return t_ret
        Call(f, args):
            # lhs flows into last argument slot
            let t_new = TyFn(map(infer, args) ++ [t_lhs], fresh_ty())
            unify(infer(env, f), t_new)
            return t_new.ret
```

The Call branch is what makes xs |> map(f) work: we treat map(f) as though xs were its final argument, and the unifier threads the element type through without intervention.

6. Records and Row Polymorphism

Record literals carry an optional row variable: {x: 1, y: 2} has type {x: int, y: int | r} where r is a fresh row variable. Field projection rec.x unifies rec with {x: a | r} and yields a. This gives us duck-typed projection that still enjoys inference:

```
fn distance(p) -> float:
    # p is inferred as {x: float, y: float | r}
    return sqrt(p.x * p.x + p.y * p.y)
```

```
distance({x: 3.0, y: 4.0})              # OK
distance({x: 3.0, y: 4.0, z: 5.0})      # also OK, r = {z: float}
```

Row unification is strictly more work than ML-style records, but the extra bookkeeping is contained: each TyRecord carries a row variable that is unified independently of its field map.

7. Gradual Fallback

When an expression is annotated any, inference treats it as a universal-quantified type variable that unifies with anything and propagates as any through the substitution. The result is a type system that is sound on fully-annotated fragments and permissive on any-annotated fragments, with a sharp boundary between the two. Callers of an any-returning function must themselves annotate or accept any, preventing silent drift.

8. Testing

The inference test suite in tests/test_v15_features.py has three tiers:

- **Unit tests for the unifier** (~40 cases): unify TyVar with TyCon, TyFn with TyFn, occurs-check failure, row-record unification, etc.
- **End-to-end inference tests** (~120 cases): parse a Lateralus source fragment, run inference, compare the reported type of a named binding against a string expectation. Example: assert_type("let x = [1,2,3]", "x", "list[int]").
- **Diagnostic tests** (~25 cases): check that inference failures produce the expected error code and position, not just any failure.

The diagnostic tier is what gives us confidence that a production regression will be caught at the same time as a correctness regression. A silent fallback to any is a correctness bug in the tests as well as in the compiler.

9. Future Work

Three extensions are on the roadmap:

- **Higher-rank types** for functions that take polymorphic callbacks (fn map_all(f: forall a. a -> a, xs)).
- **GADTs** for parser-combinator-style code where match arms refine type indices.
- **Effect rows** co-existing with the record rows, giving us IO/async/throws tracking at the same site as the field types.

Each of these is a principled extension; none require rewriting the core unifier. We expect them over the 1.6 and 1.7 releases respectively.

10. Conclusion

Lateralus 1.5 delivers an Algorithm W variant tuned for pipeline-heavy code, with row-polymorphic records and a well-defined gradual escape hatch. The implementation fits in under a thousand lines of Python and passes 185 tests covering the inferable fragment, the gradual boundary, and the diagnostic layer. It is the same implementation we rely on in the editor plugins and the lateralus check subcommand, giving us one source of truth for what the language means.