Property-Based Testing as a Compiler Pass

A design note on lightweight randomized testing in Lateralus 0.5

Lateralus Language

bad-antics · April 2026 · Lateralus Language Research

ABSTRACT Property-based testing is widely acknowledged as a superior complement to example-based unit tests, yet adoption lags because of the boilerplate required to set up generators, shrinkers, and driver functions. We describe Lateralus' approach, in which property testing is implemented as a compiler analysis pass rather than a library. The pass inspects #[property]-annotated functions, reads argument types from the type checker, and synthesizes a shrinking driver function marked #[test]. The result is near-zero user-facing boilerplate, type-aware generator selection, and natural composition with other passes such as observability instrumentation. We report implementation experience from the Lateralus compiler, outline the generator-type algebra, and discuss why the compiler-pass design is particularly well-suited to pipeline-native languages.

1. Introduction

Since QuickCheck's introduction in 2000, the value of property-based testing has been well established: by sampling a distribution of inputs rather than pinning down individual examples, tests exercise invariants across a broad region of the input space and routinely surface corner cases that example-based suites miss. Adoption, however, has been uneven. A survey of open-source Haskell projects found property tests in 9% of packages despite QuickCheck's cultural prominence; the corresponding Python figure (Hypothesis) sits at 2–4%. The common refrain from practitioners is not that the idea is hard, but that the plumbing is: defining a generator for a record type, registering shrinkers, wiring RNG state through the test framework, and ensuring reproducibility all add up.

This paper describes an alternative design implemented in Lateralus 0.5. Rather than shipping a library, the compiler exposes a single attribute, #[property], which triggers an analysis pass that synthesizes the property driver from the function's type signature. The result is that the user-visible surface of the feature is one attribute and zero glue code.

2. Design Goals

We imposed four constraints on the design:

- **No user-written generators for stdlib types.** If the function takes [Int], the compiler must know how to generate one.
- **No runtime tax when properties are disabled.** A release build compiled without --test emits nothing from the pass.
- **Composability with other passes.** The observability and capability-tracking passes must see the generated drivers and treat them uniformly.
- **Deterministic reproduction.** A failing property reports a seed sufficient to reproduce the exact counterexample.

3. The User-Facing Surface

The entire public API is one attribute, applied to any Boolean- or Result-returning function:

```
#[property(runs = 500, seed = 0xC0FFEE)]
fn reverse_twice_is_identity(xs: [Int]) -> Bool {
    xs |> reverse() |> reverse() == xs
}
```

Tunable attributes are runs (default 100), max_shrink (default 128), and seed (default: derived from LATERALUS_TEST_SEED or a CSPRNG). A property function may return either Bool or Result<(), E>; the pass selects the appropriate assertion shape at codegen time based on the type checker's output.

4. Pass Architecture

The pass runs after type inference and before ownership analysis, in the position traditionally occupied by macro expansion in other compilers. It operates in three phases:

4.1 Discovery

The pass walks module.items and filters to function declarations carrying an #[property] attribute. For each, it reads the argument types from the attached type environment and constructs a Property record capturing the function name, argument type vector, and attribute-derived knobs.

4.2 Driver Synthesis

For each Property, the pass emits a zero-argument function __prop_<name>() marked #[test]. The generated body follows a fixed template:

```
fn __prop_reverse_twice_is_identity() {
    let rng = std::rand::Rng::seeded(0xC0FFEE)
    let failures = []
    for _ in 0..500 { __prop_one(rng, failures) }
    if len(failures) > 0 {
        panic("property failed after shrinking: {failures[0]}")
    }
}
```

The __prop_one helper generates a sample for each argument type, invokes the property function, and on failure drives the shrinker. Because the generated function is a normal #[test], it appears in the existing test runner output without any framework changes.

4.3 Generator Resolution

For each argument type T, the pass synthesizes a call to std::test::arbitrary::<T>(rng). For stdlib types, arbitrary is implemented by ordinary function overloading on type; for user-defined types, the derive attribute #[derive(Arbitrary)] emits an implementation based on the type's field types. If no implementation is in scope, the compiler emits a diagnostic pointing at the property declaration, not at an obscure trait-resolution error.

5. Composition With Other Passes

Because generated drivers are ordinary #[test] functions at the AST level, they are naturally visible to subsequent passes. Two examples:

5.1 Observability

The #[traced] pass (Emit: compiler/codegen/otel_emit.ltl) wraps every test body in an OTEL span. Property tests therefore appear as spans in trace backends, allowing runtime distribution analysis (which inputs took longest, which triggered failure) without additional instrumentation.

5.2 Capability Tracking

Lateralus' capability-based security pass rejects compilation if a test function touches a capability (e.g., network I/O) not declared in the test's allowlist. Property drivers inherit this check mechanically, ruling out classes of accidental integration-in-unit-tests before they reach CI.

6. Shrinking

Shrinking in our implementation is recursive and type-directed, modeled on Hypothesis' "internal shrinker": the RNG is replayed with a minimized byte-stream that reproduces the failing example, then the stream is progressively simplified (bytes zeroed or removed) while the property still fails. This decouples shrinking from individual generator functions, which dramatically simplifies the derive macro for Arbitrary. The budget max_shrink = 128 was chosen empirically on the Lateralus stdlib test suite; shrinking typically converges within 20–40 steps for list-structured types.

7. Reproduction and CI

On failure, the pass emits a diagnostic of the form:

```
property failed after shrinking:
    reverse_twice_is_identity(xs = [0, 1])
    seed = 0xC0FFEE, shrink_step = 34
    reproduce: LATERALUS_TEST_SEED=0xC0FFEE lateralus test reverse_twice_is_identity
```

The seed is always captured, even when derived from the CSPRNG, ensuring that flaky tests can always be reproduced locally from a CI log line. The pass additionally writes a stable regression file at .lateralus/regressions/<property>.seeds, which is replayed on every subsequent test invocation.

8. Implementation Cost

The entire pass is approximately 180 lines of Lateralus, with an additional 60 lines of AST-builder helpers and 240 lines of stdlib Arbitrary implementations (covering Int, Float, Bool, String, [T], Map<K,V>, Option<T>, Result<T,E>, and tuples up to arity 8). The compile-time overhead on a 10k-line test module is under 4% of total compile time.

9. Discussion

Two design decisions merit discussion. First, we chose a compiler pass over macros despite the latter being the more conventional solution. The determining factor was type visibility: a macro operating on syntactic trees cannot query the type environment, which would have forced users to annotate argument types even when the compiler already knew them. Second, we chose to emit drivers as #[test] functions rather than inventing a new test category; this reuses existing test infrastructure (the runner, the filter syntax, the JSON reporter) at no additional cost.

Pipeline-native languages are particularly well-served by this design because pipeline stages are individually typed and individually named, which means property tests over pipelines can target stages independently. A pipeline xs |> filter(p) |> sort() admits three distinct properties (over p, over sort's output, and over the composition) without restructuring the function.

10. Conclusion

Lifting property-based testing from library to compiler pass removes the primary adoption barrier — boilerplate — at negligible cost to the compiler and at no cost to users who do not opt in. The same approach generalizes naturally to other forms of randomized verification: fuzzing harnesses, concolic-execution drivers, and contract-based interface tests are all candidates for similar treatment. We believe language-level support for verification techniques is a productive direction for general-purpose languages and invite experimentation.