Gradual Typing in Lateralus v1.5

The dyn type, consistency relation, and type-safe interop with dynamic data

Lateralus Language

bad-antics · November 2024 · Lateralus Language Research

ABSTRACT Lateralus v1.5 introduces gradual typing via the dyn type: a value of type dyn can hold any Lateralus value and can be passed anywhere without a type error. Type casts from dyn to a concrete type are checked at runtime. The gradual type system is based on the consistency relation of Siek & Taha (2006): a dyn value is consistent with any concrete type, and the consistency relation replaces equality in the type rules for expressions involving dyn. This paper specifies the gradual type system, explains its interaction with pipeline operators and error propagation, and evaluates the runtime overhead of cast checks.

1. Motivation: Interop with Dynamic Data

Fully static type systems excel at enforcing invariants over data the programmer controls. They are less ergonomic for data whose structure is known only at runtime: JSON API responses, configuration files, plugin return values, and script-evaluated expressions.

Rather than requiring the programmer to write a fully-typed decoder before using any external data, Lateralus v1.5 allows external data to enter the program as dyn and be cast to concrete types at the boundary where they are used. This shifts type checking from compile time to runtime for the dynamic portion, while preserving full static checking everywhere else.

```
// Parse a JSON blob — value is dyn
let data: dyn = json::parse(response_body)?;

// Cast to a concrete type at the use site
let name: str = data["name"].cast::<str>()?;
let age: u32  = data["age"].cast::<u32>()?;

// From here, name and age are statically typed
```

2. The Consistency Relation

Two types A and B are consistent (written A ~ B) if they can appear in place of each other in a context that expects either. The consistency relation is reflexive and symmetric but not transitive:

```
-- Reflexivity
A ~ A     for any type A

-- dyn is consistent with everything
A ~ dyn   for any type A
dyn ~ A   for any type A

-- NOT transitive:
str ~ dyn ~ u32  does NOT imply str ~ u32
```

The consistency relation replaces equality in the type rule for function application: if a function expects type A and the argument has type B where A ~ B, the application is type-correct at compile time. If B = dyn, the compiler inserts a runtime cast check.

```
-- Gradual function application
Gamma |- f : A -> C    Gamma |- x : B    A ~ B
-------------------------------------------------
         Gamma |- f(x) : C  (+ cast check if B = dyn)
```

3. dyn in Pipeline Stages

A pipeline stage that accepts dyn can receive any value from the previous stage. A stage that produces dyn can feed any downstream stage. The consistency relation handles the type rules:

```
let result = json_data     // dyn
    |> extract_fields      // dyn -> dyn (structural extraction)
    |?> cast_to_user       // dyn -> Result<User, CastError>
    |>  process_user       // User -> ProcessedUser (fully typed)
```

The pipeline transitions from dynamic to static at the cast_to_user stage. After that stage, the type is statically known as User and all subsequent stages are checked at compile time with no runtime overhead.

3.1 Dynamic Pipelines

A fully dynamic pipeline (all stages accept and return dyn) behaves like a dynamically typed language for that section of code. The compiler inserts cast checks at every stage boundary. This is useful for scripting-like DSLs or for pipeline stages loaded from plugins at runtime.

4. Cast Semantics

A cast from dyn to type T checks whether the runtime tag of the value matches T. If it matches, the cast succeeds and returns Ok(t). If it does not, the cast returns Err(CastError { expected: T, actual: tag }).

```
fn cast<T>(v: dyn) -> Result<T, CastError> {
    if v.tag() == TypeId::of::<T>() {
        Ok(unsafe { v.into_inner::<T>() })
    } else {
        Err(CastError { expected: TypeId::of::<T>(), actual: v.tag() })
    }
}
```

The cast always returns Result, making it compatible with |?>. A pipeline that casts at every stage and short-circuits on failure produces a clean error message that includes the expected and actual types at the stage where the cast failed.

5. The Blame System

In gradual type systems, a cast failure at runtime may be caused by a type error at a definition site far from the cast. The blame system tracks which part of the code is responsible for a cast failure: the producer that put the wrong type into a dyn container, or the consumer that expected a type the producer cannot provide.

Lateralus implements a lightweight blame system: every dyn value carries a source location tag that records where it was created. When a cast fails, the error includes both the cast location (the consumer) and the creation location (the producer).

```
error[E0601]: cast failure
  --> src/handler.lt:42:8
   |
42 |     |?> cast_to_user  // expected: User
   |         ^^^^^^^^^^^ cast to User failed
   |
note: dyn value was created here with type XmlDoc
  --> src/parser.lt:18:5
18 |     let data: dyn = parse_xml(body)
```

```
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ XmlDoc introduced as dyn
```

6. Performance: Cast Check Overhead

Each runtime cast check is a pointer comparison between the value's type tag and the expected TypeId. On a modern processor, this is a single branch instruction with a predictable outcome for hot paths.

```
Benchmark                    Cast overhead   Notes
-----------------------------------------------------
Hot path (100% hit rate)       < 0.5 ns     branch predictor
Mixed (80% hit rate)           ~1.2 ns      occasional mispredict
Cold path (0% hit rate)        ~3.5 ns      type error, rare
Dynamic pipeline (all dyn)    ~10 ns/stage  tag check at every boundary
```

The cast overhead is negligible for hot paths. For fully dynamic pipelines, the 10 ns/stage overhead accumulates: a 10-stage dynamic pipeline adds ~100 ns compared to a fully static one. This is acceptable for interop code but argues against using dyn in performance-critical inner loops.

7. Migration Path: dyn to Static

The gradual type system is intended as a migration tool, not a permanent state. Code that begins with dyn boundaries should progressively narrow them as the data schema is understood and stabilized.

The compiler provides a --find-dyn flag that reports all dyn expressions in a codebase, grouped by the number of times each one is cast to a specific type. When a dyn value is always cast to T, the compiler suggests replacing dyn with T at the source.

8. Conclusion

Gradual typing via the dyn type and consistency relation extends Lateralus's static type system to accommodate dynamic data without abandoning static checking for fully-typed code. The blame system provides actionable runtime error messages. The cast overhead is negligible on hot paths.

The gradual type system is compatible with all four pipeline operators: dyn can flow through total, error-propagating, async, and fan-out stages with appropriate cast checks inserted automatically. This makes the migration from dynamic to static typing a matter of adding type annotations, not rewriting pipeline structure.