Pattern Matching and ADTs in Lateralus

Sum types, exhaustiveness checking, and guard expressions

Lateralus Language

bad-antics · December 2024 · Lateralus Language Research

ABSTRACT Algebraic data types (ADTs) and pattern matching are the primary tools for modeling domain concepts in Lateralus. This paper describes the enum declaration form for sum types, the match expression for exhaustive pattern matching, the guard expression extension, and how ADTs integrate with the four pipeline operators. We pay particular attention to exhaustiveness checking — the compiler analysis that ensures every possible value of an ADT is handled — and to the interaction between patterns and the error propagation operator.

1. Algebraic Data Types

An algebraic data type in Lateralus is declared with the enum keyword. Each variant can carry zero or more fields:

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: f64, height: f64 },
    Triangle { base: f64, height: f64 },
    Point,   // unit variant: no fields
}

enum ParseResult<T> {
    Ok(T),
    Err(ParseError),
    Incomplete { needed: usize },
}
```

ADTs are the canonical way to represent data that can take one of several distinct forms. The compiler tracks which variant a value has at compile time and flags any code that accesses a variant's fields without first matching to determine which variant is present.

2. Pattern Matching: The match Expression

The match expression destructures a value against a list of patterns. The first matching pattern wins; its bound variables are in scope in the body:

```
let area = match shape {
    Shape::Circle { radius }       => math::pi * radius * radius,
    Shape::Rectangle { width, height } => width * height,
    Shape::Triangle { base, height }   => 0.5 * base * height,
    Shape::Point                   => 0.0,
}
```

Patterns can bind values, ignore values with _, and use range patterns for numeric types.

2.1 Nested Patterns

Patterns can be arbitrarily nested: a pattern for a Result<Shape, Error> can simultaneously match the outer variant and destructure the inner value:

```
let result = match parse_shape(input) {
    Ok(Shape::Circle { radius }) if radius > 0.0 => process_circle(radius),
    Ok(Shape::Circle { .. })                     => Err("zero radius"),
    Ok(other)                                    => process_other(other),
    Err(e)                                       => Err(e.to_string()),
}
```

3. Exhaustiveness Checking

The compiler checks that every match expression covers all possible values of the matched type. A match is exhaustive if for every possible constructor and field combination, at least one arm matches.

The exhaustiveness checker uses the 'useful' algorithm from Maranget (2007): a matrix of patterns is checked for usefulness, where a pattern is useful if it covers at least one case not covered by earlier patterns.

```
// Compiler error: non-exhaustive match
let area = match shape {
    Shape::Circle { radius }    => math::pi * radius * radius,
    Shape::Rectangle { width, height } => width * height,
    // missing: Triangle and Point
};
// error[E0301]: non-exhaustive match on Shape
// not covered: Triangle { .. }, Point
```

The error message lists exactly which constructors are missing, making the fix obvious. Adding a wildcard arm (_ => ...) makes the match exhaustive but loses the compile-time guarantee that new variants will be handled when added.

3.1 Sealed Enums and Exhaustiveness

A sealed enum cannot be extended by downstream code, guaranteeing that the set of variants is fixed. The exhaustiveness checker relies on this property: if an enum is not sealed, the checker cannot prove exhaustiveness for code outside the defining module.

4. Guard Expressions

A guard is a boolean expression attached to a match arm that further refines which values the arm matches. If the guard evaluates to false, the arm is skipped and the next arm is tried:

```
let classification = match n {
    n if n < 0    => "negative",
    0             => "zero",
    n if n < 100  => "small positive",
    n if n < 1000 => "medium positive",
    _             => "large positive",
}
```

Guards interact with exhaustiveness checking: a match arm with a guard is not considered to fully cover the pattern it matches. A match that covers all constructors but with guards on every arm is still non-exhaustive.

5. ADTs and Pipeline Operators

The |?> operator's error type is a special case of ADTs: Result<T, E> is the ADT enum Result { Ok(T), Err(E) }. The error operator short-circuits on Err, which is equivalent to matching the Err arm and returning early.

General ADTs can be used as pipeline values. A function that transforms a Shape to another Shape is a valid |> stage. A function that fails on degenerate cases (zero-radius circles) can return Result<Shape, ShapeError> and be used as a |?> stage.

```
fn normalize_shape(s: Shape) -> Result<Shape, ShapeError> {
    match s {
```

```
        Shape::Circle { radius } if radius <= 0.0 =>
            Err(ShapeError::DegenerateRadius),
        Shape::Rectangle { width, height } if width <= 0.0 || height <= 0.0 =>
            Err(ShapeError::DegenerateDimension),
        valid => Ok(valid),
    }
}

let area = input_shape
    |?> normalize_shape
    |>  compute_area
```

6. Or-Patterns and Pattern Binding

Multiple patterns can be combined with | to match any of several cases with the same arm body. All branches of an or-pattern must bind the same set of variables with the same types:

```
let is_degenerate = match shape {
    Shape::Circle { radius: r }   if r <= 0.0 => true,
    Shape::Rectangle { width: w, .. } if w <= 0.0 => true,
    Shape::Rectangle { height: h, .. } if h <= 0.0 => true,
    _ => false,
}
```

Or-patterns at the top level of a match arm are supported; nested or-patterns (inside a tuple or struct pattern) are also supported and interact with exhaustiveness checking as expected.

7. ADTs in the Type System

ADT variants are types: Shape::Circle is a type with a single constructor and fields { radius: f64 }. This makes the type system consistent: pattern matching is structural decomposition, and the exhaustiveness checker works over the same type information that the rest of the type system uses.

Generic ADTs follow the same rules. A Result<T, E> is exhaustively matched by Ok(t) and Err(e) for any T and E. The type parameters are inferred from context.

8. Practical Patterns

Two ADT patterns that appear frequently in Lateralus codebases:

8.1 The Newtype Pattern

Wrapping a primitive in a unit-variant enum to distinguish semantically different values with the same underlying type:

```
enum UserId(u64);
enum SessionId(u64);

// Compiler prevents swapping UserId and SessionId even though
// both are u64 underneath
fn find_user(id: UserId) -> Option<User> { ... }
```

8.2 The Builder ADT Pattern

Using an ADT to represent the states of a builder, making it impossible to call build() without providing required fields:

```
enum RequestBuilder {
```

```
    Empty,
    WithUrl { url: str },
    WithUrlAndMethod { url: str, method: HttpMethod },
}

// Only WithUrlAndMethod can be built
fn build(b: RequestBuilder::WithUrlAndMethod) -> Request { ... }
```