Pattern Matching and Algebraic Data Types in Lateralus

bad-antics | May 2024 | Language Design

Abstract

This paper presents the design and implementation of algebraic data types (ADTs) and pattern matching in the Lateralus programming language. We describe sum types, product types, exhaustiveness checking, nested patterns, guard clauses, and the integration of pattern matching with pipeline operators.

1 Introduction

Algebraic data types and pattern matching are foundational features of the Lateralus type system, enabling developers to model complex data structures with precision and decompose them with clarity. ADTs combine sum types (tagged unions) and product types (records) into a unified framework that integrates naturally with the pipeline-native design philosophy of the language.

This paper presents the design and implementation of ADTs and pattern matching in Lateralus. We describe the syntax for defining sum and product types, the pattern matching syntax with support for nested patterns and guard clauses, the exhaustiveness checker that ensures all cases are handled at compile time, and the compilation strategy that generates efficient native code from match expressions.

Pattern matching interacts with Lateralus pipelines through the match expression, which can appear as a pipeline stage. This integration allows developers to write data transformation pipelines that branch on the shape of their data, combining the expressiveness of pattern matching with the composability of pipelines. The result is a language where complex data processing is both type-safe and elegant.

2 Sum Types

Sum types in Lateralus are declared using the enum keyword. An enum defines a closed set of variants, each of which may carry zero or more fields of associated data. The type system treats each variant as a constructor function that produces a value of the enum type when applied to the appropriate arguments.

```
enum Shape {
    Circle(Float),
    Rectangle(Float, Float),
    Triangle(Float, Float, Float),
    Point,
}

enum Option<T> {
    Some(T),
```

```
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Enum variants are first-class values. The variant Circle is a function of type Float -> Shape that can be passed to higher-order functions, stored in variables, and used in pipeline stages. This functional treatment of constructors enables concise data construction patterns such as mapping a list of radii to a list of circles.

Generic enums use type parameters that are specified at the definition site and inferred at use sites. The Option<T> type is the canonical example: it represents a value that may or may not be present, replacing null references with a type-safe alternative that the compiler can check exhaustively.

Recursive enums are supported through explicit boxing. A variant that contains a value of its own enum type must wrap that value in a Box<T> to ensure the type has a finite size. The compiler detects unboxed recursive types and reports a clear error message suggesting the fix.

3 Product Types

Product types in Lateralus are declared using the struct keyword. A struct defines a record with named fields, each of which has a type. Structs can be generic, parameterized by type variables that appear in their field types. Tuple structs provide an alternative syntax with positional fields for simple wrappers.

```
struct Point {
    x: Float,
    y: Float,
}

struct Color(u8, u8, u8);

struct Pair<A, B> {
    first: A,
    second: B,
}
```

Struct fields are private by default and can be made public with the pub modifier. This encapsulation allows the struct's implementation to change without affecting code that uses the struct through its public API. Pattern matching on structs with private fields is restricted to the module that defines the struct.

Structs support derived implementations of common traits. The derive attribute automatically generates implementations for traits like Eq, Hash, Debug, and Clone based on the struct's field

types. Derived implementations reduce boilerplate and ensure consistency between the struct's definition and its trait implementations.

4 Pattern Matching Fundamentals

The match expression evaluates a scrutinee against a sequence of arms. Each arm consists of a pattern and a body expression. The match expression evaluates the scrutinee, then tests each arm's pattern in order. The body of the first matching arm is evaluated and its result becomes the value of the match expression.

```
fn area(shape: Shape) -> Float {
    match shape {
        Circle(r) => 3.14159 * r * r,
        Rectangle(w, h) => w * h,
        Triangle(a, b, c) => {
            let s = (a + b + c) / 2.0;
            sqrt(s * (s-a) * (s-b) * (s-c))
        }
        Point => 0.0,
    }
}
```

Patterns support several forms: literal patterns that match exact values, variable patterns that bind the matched value to a name, constructor patterns that decompose enum variants into their fields, struct patterns that match named fields, wildcard patterns that match anything without binding, and tuple patterns that match positional elements.

Guard clauses add boolean conditions to patterns, allowing more precise discrimination than structural patterns alone. A guarded pattern matches only when both the structural pattern matches and the guard expression evaluates to true. Guards have access to all variables bound by the pattern, enabling expressive conditions.

```
fn classify(n: Int) -> String {
    match n {
        0 => "zero",
        x if x > 0 && x < 10 => "small positive",
        x if x >= 10 => "large positive",
        x if x > -10 => "small negative",
        _ => "large negative",
    }
}
```

5 Exhaustiveness Checking

The exhaustiveness checker verifies that a match expression covers all possible values of the scrutinee type. For enum types, every variant must be matched by at least one arm. For numeric types, a wildcard or variable pattern is required. For struct types, all fields must be accounted for in

the pattern.

The algorithm computes the set of uncovered patterns: values that no arm matches. If this set is non-empty, the compiler reports an error listing example values that are not covered. The error messages include concrete example values that help the developer understand what cases they missed.

Usefulness checking is the dual of exhaustiveness. It verifies that each arm is reachable: there exists at least one value that matches the arm and does not match any earlier arm. Unreachable arms indicate dead code and are reported as warnings. Together, exhaustiveness and usefulness ensure that every arm contributes to the match.

Guard clauses are treated conservatively in exhaustiveness analysis. The checker assumes that any guard may evaluate to false, so a guarded arm does not contribute to exhaustiveness coverage. This conservative approach may produce false positive exhaustiveness errors, but it never misses a genuinely incomplete match.

6 Nested and Compound Patterns

Patterns can be nested to arbitrary depth, matching the structure of nested data types. A nested pattern decomposes values layer by layer, binding variables at any depth. This capability is essential for working with recursive data structures like expression trees and linked lists.

```
enum Expr {
    Lit(Int),
    Add(Box<Expr>, Box<Expr>),
    Mul(Box<Expr>, Box<Expr>),
    Neg(Box<Expr>),
}

fn simplify(e: Expr) -> Expr {
    match e {
        Add(box Lit(0), box r) => simplify(r),
        Add(box l, box Lit(0)) => simplify(l),
        Mul(box Lit(0), _) => Lit(0),
        Mul(box Lit(1), box r) => simplify(r),
        Neg(box Neg(box inner)) => simplify(inner),
        other => other,
    }
}
```

Or-patterns allow a single arm to match multiple alternatives. The syntax p1 | p2 matches if either pattern matches. All alternatives must bind the same variables with the same types, ensuring the arm body can use bound variables regardless of which alternative matched. Or-patterns reduce code duplication in match expressions.

As-patterns bind the entire matched value to a variable while also decomposing it. The syntax x @

Pattern binds the matched value to x while requiring it to match Pattern. This is useful when the arm body needs both the decomposed parts and the whole value, avoiding the need to reconstruct the value from its parts.

Slice patterns match arrays and vectors by their elements. The pattern [first, second, ..rest] binds the first two elements and collects the remaining elements into a slice. Slice patterns support matching from the end with [..init, last] and matching a specific length with [a, b, c] for three-element arrays.

7 Pipeline Integration

Pattern matching integrates with pipelines through the match_map and match_filter combinators. The match_map combinator applies pattern matching within a map stage, transforming each element based on its variant. The match_filter combinator uses patterns to select elements, keeping only those that match.

```
// Pattern matching in a pipeline
shapes
    |> match_map {
        Circle(r) => r * r * 3.14159,
        Rectangle(w, h) => w * h,
        Triangle(a, b, c) => {
            let s = (a + b + c) / 2.0;
            sqrt(s * (s-a) * (s-b) * (s-c))
        }
        Point => 0.0,
    }
    |> filter(|area| area > 10.0)
    |> sort()
```

The pipeline integration preserves exhaustiveness checking. The compiler verifies that match_map patterns are exhaustive, ensuring every element in the pipeline is handled. This prevents runtime failures from unmatched patterns in the middle of a data processing pipeline, which would be difficult to debug in production.

The match_filter combinator keeps only elements matching a given pattern, discarding the rest. Unlike match_map which must be exhaustive, match_filter is inherently partial: unmatched elements are simply filtered out. This provides a type-safe alternative to runtime type checks in pipeline processing.

8 Type Inference with ADTs

Type inference for ADTs uses a constraint-based algorithm. When a pattern matches a constructor, the algorithm generates constraints linking the scrutinee type to the constructor's parent enum and the bound variables to the constructor's field types. These constraints propagate bidirectionally through the match expression.

Generic ADTs require unification of type variables across arms. When multiple arms bind variables of the same generic type at different instantiations, the algorithm detects the conflict and reports a type error. Error messages show the conflicting instantiations and the arms where they occur.

The interaction between type inference and exhaustiveness checking requires careful ordering. Type inference runs first to determine the scrutinee type, then exhaustiveness checking runs on the fully-typed match expression. This ordering ensures exhaustiveness analysis has complete type information for variant enumeration.

9 Compilation to Decision Trees

Match expressions are compiled to decision trees that minimize runtime tests. The compiler analyzes patterns to find the most discriminating column and splits the match into sub-problems. Each leaf of the decision tree corresponds to an arm body, and each internal node tests one component of the scrutinee.

The decision tree is lowered to a sequence of conditional branches and field accesses. For enum types, the first test switches on the variant tag, followed by field extraction. This strategy produces code equivalent in efficiency to hand-written switch statements, with the additional benefit of verified exhaustiveness.

The compiler applies several optimizations: sharing detection merges identical subtrees, lookup tables replace decision trees for dense literal patterns, and single-arm matches are simplified to conditional checks. These optimizations keep the generated code compact and cache-friendly.

10 Conclusion

Pattern matching and ADTs in Lateralus provide a powerful and safe mechanism for working with structured data. The integration with pipelines through match_map and match_filter enables expressive data processing that branches on data shape while maintaining exhaustiveness guarantees at compile time.

2.1 Variant Constructors as Functions

Because enum variants are first-class constructors, they can be used anywhere a function is expected. The variant Circle has type Float -> Shape, making it directly usable as an argument to map, filter, and other higher-order functions. This eliminates the need for wrapper closures when constructing values in pipelines.

Constructor functions participate in type inference like any other function. When Circle is passed to a generic function expecting T -> Shape, the type parameter T is inferred as Float from Circle's signature. This inference works across pipeline stages, allowing the type checker to verify entire data transformation chains.

Variant constructors support partial application when they take multiple arguments. The variant

Rectangle can be partially applied as Rectangle(5.0) to create a function Float -> Shape that produces rectangles with a fixed width. Partial application of constructors enables point-free style in pipeline expressions.

The functional treatment of constructors extends to pattern matching. A constructor can be used as a function in a map stage to wrap values, and as a pattern in a match expression to unwrap them. This symmetry between construction and destruction is a hallmark of algebraic data types.

Constructor functions are monomorphized during compilation, meaning each instantiation of a generic constructor generates specialized code. The constructor Some for Option<Int> generates different code than Some for Option<String>, ensuring optimal performance without boxing overhead.

Named constructors provide self-documenting code. Unlike tuple variants where the meaning of each field must be inferred from context, named-field variants make the purpose of each field explicit. The compiler supports both syntaxes interchangeably, allowing developers to choose the style that best communicates intent.

The compiler generates efficient constructor code that initializes the variant tag and copies field data in a single operation. For small variants that fit in registers, construction is a single instruction. For larger variants, the compiler uses memcpy-style initialization that is optimized by the backend.

Constructor visibility follows the same rules as field visibility. A public enum with private variant fields can be constructed only within the defining module, but can be pattern-matched from outside. This allows the module to maintain invariants on the variant's fields while still supporting external pattern matching.

Variant constructors interact with the trait system through the From and Into conversion traits. Implementing From<Float> for Shape allows implicit conversion from Float to Shape through the Circle variant, enabling ergonomic API design where callers can pass floats directly where shapes are expected.

The constant evaluation engine can evaluate constructor expressions at compile time, enabling const-initialized enum values. A const value of type Shape can be created by calling Circle(1.0) in a const context, and the resulting value is embedded directly in the program binary without runtime initialization.

3.1 Struct Patterns and Field Access

Struct patterns match values by their field names and values. The pattern Point { x, y } matches a Point and binds its fields to local variables with the same names. Field patterns can be reordered: Point { y, x } is equivalent to Point { x, y }. This flexibility makes code resilient to field reordering in the struct definition.

Partial struct patterns use the .. syntax to ignore unmatched fields. The pattern Point { x, .. } matches any Point and binds only the x field. Rest patterns are essential for forward compatibility: adding a

new field to a struct does not break pattern matches that use rest syntax.

Struct patterns can nest other patterns for their fields. The pattern Rect { origin: Point { x, .. }, .. } destructures a Rect and its nested Point, binding only the x coordinate of the origin. Nested struct patterns enable deep access into complex data structures in a single pattern.

Named field binding allows renaming: Point { x: horizontal, y: vertical } binds the x field to the variable horizontal. This is useful when the field name conflicts with a local variable or when a more descriptive name improves readability in the arm body.

Struct patterns interact with visibility: patterns can only match fields that are visible at the match site. Matching a struct with private fields requires the rest pattern .. to acknowledge the existence of fields that cannot be accessed. The compiler reports which fields are private in its error messages.

The compiler verifies that struct patterns mention each field at most once and that all mentioned fields exist in the struct definition. Duplicate fields in a pattern are a compile error. Unknown fields generate an error with a suggestion for the closest matching field name, helping catch typos.

Struct patterns support literal field values for filtering. The pattern Config { debug: true, .. } matches only Config values where the debug field is true. Literal field patterns combine naturally with guard clauses for complex filtering conditions on struct values.

Mutable bindings in struct patterns use the mut keyword: Point { mut x, y } binds x as a mutable variable. Mutable bindings allow the arm body to modify the bound value without affecting the original struct, since pattern matching creates copies of matched fields for Copy types.

Struct patterns are compiled to a sequence of field accesses followed by tests for literal fields and recursive pattern matching for nested fields. The compiler orders the field accesses to minimize memory latency, accessing fields at increasing offsets when possible to take advantage of hardware prefetching.

Default field values in struct definitions interact with patterns by providing fallback values for fields not specified during construction. A pattern always matches against the actual field value, including defaults. The pattern Config { verbose: true, .. } will match only when verbose is explicitly true, not when it was set by a default.

5.1 Exhaustiveness Algorithm Details

The exhaustiveness algorithm uses the concept of a pattern matrix: a two-dimensional structure where each row represents an arm's pattern decomposed into columns corresponding to the scrutinee's components. The algorithm recursively processes the matrix, specializing it by constructor and computing the uncovered set.

Specialization removes a constructor from the matrix by keeping only rows that match that constructor and removing the constructor's column. The remaining columns contain the patterns for the constructor's fields. This process continues recursively until the matrix is empty or all constructors have been processed.

The default matrix contains rows that match any constructor not explicitly listed. These rows come from wildcard and variable patterns. The default matrix is used to check coverage of constructors not mentioned in any arm, which is important for open types and types with many variants.

For types with a large number of variants (such as integer types), the algorithm switches from variant enumeration to a completeness check based on the presence of wildcard patterns. If any arm has a wildcard pattern at the relevant column, all values of that type are covered.

The algorithm generates witness patterns: concrete values that demonstrate uncovered cases. When exhaustiveness fails, the witness pattern is displayed in the error message. For nested types, the witness pattern can be complex, such as Some(Rectangle(_, 0.0)) indicating that a specific nested case is not handled.

Or-patterns are expanded into separate rows before exhaustiveness checking. The pattern Circle(r) | Point is split into two rows: one for Circle(r) and one for Point. This expansion ensures that each alternative contributes independently to coverage, and the or-pattern's semantics are correctly handled.

Guard clauses affect exhaustiveness by making their arms conditionally reachable. An arm with a guard is treated as covering its pattern only when the guard is true. Since guards can have arbitrary boolean expressions, the checker conservatively assumes they may be false, potentially leaving cases uncovered.

The exhaustiveness checker handles recursive types by limiting the depth of witness pattern generation. For a recursive enum like List, the checker generates witnesses up to a configurable depth, producing patterns like Cons(_, Cons(_, Nil)) rather than exploring infinite nesting. The depth limit is sufficient for practical error reporting.

Performance optimizations in the exhaustiveness checker include caching the covered set for each matrix position and early termination when all constructors are covered. These optimizations keep the checker fast even for match expressions with many arms and deeply nested patterns, which is important for large codebases.

The exhaustiveness checker produces both hard errors (incomplete matches) and soft warnings (redundant arms). A redundant arm is one that can never match because earlier arms cover all its cases. Redundant arms are often the result of refactoring and indicate code that should be cleaned up.

6.1 Advanced Pattern Forms

Range patterns match values within a numeric range. The pattern 1..=10 matches integers from 1 to 10 inclusive. Range patterns support both inclusive (..=) and exclusive (..) bounds and work with integer, character, and float types. They are particularly useful for classifying numeric values into categories.

Reference patterns dereference a reference and match the value behind it. The pattern &x matches

a reference and binds x to the referenced value. Reference patterns are used when matching function arguments that are borrowed, allowing the pattern to access the underlying value without explicit dereferencing.

Box patterns match heap-allocated values through their Box wrapper. The pattern box x matches a Box<T> and binds x to the contained value of type T. Box patterns are essential for matching recursive data structures where variants contain boxed self-references.

Binding modes allow patterns to automatically determine whether to bind by value, by reference, or by mutable reference based on the scrutinee's type. When matching a &Point, the pattern Point { x, y } automatically binds x and y as references. This ergonomic feature reduces the need for explicit reference patterns.

Const patterns use named constants as patterns. A pattern that names a const value matches when the scrutinee equals that value. Const patterns provide semantic names for magic numbers and are checked at compile time to ensure the constant's type matches the scrutinee's type.

Array patterns match fixed-size arrays by their elements. The pattern [a, b, c] matches a three-element array and binds each element. Array patterns support rest syntax: [first, ..middle, last] binds the first and last elements and collects the middle elements into a slice.

Tuple patterns match tuples by position. The pattern (x, y, z) matches a three-element tuple and binds each component. Tuple patterns nest naturally within other patterns: (Some(x), None) matches a pair where the first element is Some and the second is None.

The compiler desugars complex pattern forms into a canonical representation before exhaustiveness checking and code generation. This desugaring simplifies the implementation by reducing the number of pattern forms that the core algorithms must handle, while preserving the full expressiveness of the surface syntax.

Pattern macros allow users to define reusable pattern abbreviations. A pattern macro expands to a pattern at compile time, enabling domain-specific pattern vocabularies. For example, a networking library might define a pattern macro for matching IP address ranges, providing a readable alternative to nested numeric patterns.

The type checker validates that each pattern form is applicable to the scrutinee's type. A constructor pattern can only be used with the corresponding enum type. A struct pattern can only be used with the corresponding struct type. Type mismatches in patterns produce clear error messages showing the expected and actual types.

7.1 Pipeline Pattern Semantics

The match_map combinator transforms each element of a pipeline using pattern matching. It is semantically equivalent to mapping a closure that contains a match expression, but syntactically more concise. The combinator is implemented as a trait method on iterators and collections, making it available throughout the standard library.

The match_filter combinator selects elements from a pipeline based on whether they match a pattern. Elements that match are kept (and optionally transformed), while non-matching elements are discarded. This combinator is particularly useful for extracting specific variants from a heterogeneous collection.

The match_flat_map combinator combines pattern matching with flattening: for each element that matches, the arm body produces a sequence of values that are concatenated into the output. Non-matching elements produce empty sequences. This combinator is useful for expanding selected variants into multiple output values.

Pipeline pattern combinators support exhaustiveness checking by default. The compiler verifies that match_map arms cover all cases, preventing runtime panics from unmatched elements. The match_filter combinator does not require exhaustiveness because its semantics explicitly handle non-matching elements by discarding them.

Pattern combinators interact with lazy evaluation. In a lazy pipeline, match_map only evaluates the pattern and arm body when the next element is requested. This means that patterns for elements beyond the requested range are never tested, providing efficient early termination for pipelines that consume only a prefix of their input.

The partition_match combinator splits a pipeline into two branches based on pattern matching. Elements matching the first pattern go to one branch, and elements matching the second pattern go to the other. Both branches are processed independently and can be rejoined later. This enables divergent processing of different data shapes.

Error handling in pipeline patterns uses the try_match combinator, which returns a Result for each element. If the pattern match succeeds, the arm body's result is wrapped in Ok. If no arm matches (for non-exhaustive matches), an Err containing the unmatched value is produced. This enables graceful error handling without panicking.

The performance of pipeline pattern combinators is equivalent to hand-written loops with match expressions. The compiler inlines the combinator implementation and fuses it with adjacent pipeline stages, eliminating the abstraction overhead. Benchmarks show zero performance penalty compared to explicit match-in-loop code.

Custom pipeline pattern combinators can be defined by implementing the PatternCombinator trait. This trait specifies how pattern matching results are combined with the pipeline's data flow. Library authors use custom combinators to provide domain-specific pattern matching operations optimized for their data structures.

Debugging pipeline patterns is supported through the inspect_match combinator, which logs each pattern match attempt and its result without affecting the pipeline's data flow. This combinator is useful during development for understanding how data flows through complex pattern matching pipelines and identifying unexpected non-matches.

8.1 Advanced Type Inference

Type inference for match expressions uses a bidirectional algorithm that combines information from the scrutinee and the arm bodies. The scrutinee provides a top-down type that constrains the patterns, while the arm bodies provide a bottom-up type that determines the match expression's result type.

When the scrutinee type is a generic enum, the inference algorithm instantiates the type parameters by unifying the pattern with the enum's definition. Matching Some(x) against Option<T> produces the constraint T = typeof(x), which propagates through the rest of the type checking process.

Existential types in GADT patterns introduce fresh type variables scoped to the arm body. When a GADT variant introduces a type equality constraint, matching that variant makes the constraint available within the arm. This enables type-safe heterogeneous containers and expression evaluators.

Type inference for or-patterns requires all alternatives to produce the same type bindings. The pattern Some(x) | None where x is bound differently in each alternative is rejected because the arm body cannot consistently use x. The error message shows which bindings differ between the alternatives.

Inference for nested patterns proceeds depth-first, generating constraints at each level of nesting. A pattern like Some((x, y)) generates constraints that T = (A, B) and x: A and y: B. These constraints are solved together with constraints from other arms to determine the most general types.

The inference algorithm handles higher-ranked types in pattern positions through skolemization. When a pattern matches a value with a universally quantified type, the quantified variable is replaced with a fresh skolem constant that must not escape the arm body. This prevents the arm from making assumptions about the concrete type.

Type annotations on patterns provide hints to the inference algorithm when automatic inference is insufficient. The pattern x: Int forces x to have type Int, overriding any inferred type. Annotations are particularly useful in complex match expressions where the inferred type would be ambiguous.

The inference algorithm produces principal types: the most general type that is consistent with all constraints. If the constraints do not have a principal solution, the algorithm reports an ambiguity error with suggestions for type annotations that would resolve the ambiguity.

Performance of the type inference algorithm is typically linear in the size of the match expression. Pathological cases with deeply nested patterns and many type variables can trigger quadratic behavior, but these cases are rare in practice. The algorithm includes a complexity limit that triggers a more detailed error message.

Inference results are cached across match expressions in the same function. When the same scrutinee type is matched multiple times (for example, in nested match expressions), the cached results avoid redundant constraint generation and solving, improving compilation speed for pattern-heavy code.

References

[1] Maranget, L. Compiling Pattern Matching to Good Decision Trees. ML Workshop, 2008.

[2] Karachalias, G. et al. GADTs Meet Their Match. ICFP, 2015.

[3] Sestoft, P. ML Pattern Match Compilation and Partial Evaluation. 1996.

[4] Pierce, B. Types and Programming Languages. MIT Press, 2002.

[5] Wadler, P. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. POPL 1987.