Alga: Algebraic Parser Composition through Monadic Design

A C++20 Template Library for Type-Safe Text Processing

Technical Report

Abstract

We present Alga, a header-only C++20 template library that models parsers as composable algebraic structures. By treating parsers as elements of monoids and leveraging monadic composition patterns, Alga provides a mathematically rigorous yet practically efficient framework for text processing. The library's key innovation lies in its uniform Optional monad pattern for error handling and its implementation of algebraic operators that follow mathematical laws. We demonstrate how template metaprogramming ensures type safety while maintaining zero-cost abstractions, making Alga suitable for both research and production environments.

1 Introduction

Parser combinators have long been recognized as an elegant approach to text processing, but existing C++ implementations often sacrifice either mathematical rigor or practical efficiency. Alga addresses this gap by providing a library where parsers are first-class algebraic objects that compose through well-defined mathematical operations.

The core insight is that many parsing operations naturally form algebraic structures—particularly monoids under various composition operations. By modeling these structures explicitly and providing a uniform interface through C++20 concepts and templates, we achieve both theoretical elegance and practical efficiency.

2 Theoretical Foundations

2.1 Parsers as Monoids

At the heart of Alga lies the recognition that parsers form monoids under concatenation. A monoid $(M, \cdot, e)$ consists of a set M, an associative binary operation $\cdot$, and an identity element e.

Definition 1 (Parser Monoid)**.** Let $\mathcal{P}$ be the set of parsers over an alphabet Σ. The concatenation operation $* : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ and the empty parser ϵ form a monoid $(\mathcal{P}, *, \epsilon)$ where:

- $(p_1 * p_2) * p_3 = p_1 * (p_2 * p_3)$ (associativity)
- $p * \epsilon = \epsilon * p = p$ (identity)

This algebraic structure is preserved in Alga's implementation, where the `lc_alpha` type represents lowercase alphabetic strings forming a free monoid:

```cpp
// Concatenation operation (monoid multiplication)
lc_alpha operator*(lc_alpha const& lhs, lc_alpha const& rhs);

// Identity element (empty string)
auto identity = make_lc_alpha("");
```

Listing 1: Monoid operations in lc_alpha

2.2 The Optional Monad Pattern

Error handling in parsing naturally fits the monadic pattern. Alga employs `std::optional<T>` as its primary monad, providing:

Definition 2 (Optional Monad)**.** The Optional monad consists of:

- A type constructor: $\mathcal{M}(T) =$ `optional<T>`
- Return (pure): $\eta : T \to \mathcal{M}(T)$
- Bind: $\gg=: \mathcal{M}(T) \times (T \to \mathcal{M}(U)) \to \mathcal{M}(U)$

satisfying the monad laws.

This pattern enables elegant composition of potentially failing operations:

```cpp
// All factory functions return optional<T>
auto word = make_lc_alpha("hello");
auto stem = word >>= [](auto w) { return stemmer(w); };

// Automatic short-circuiting on failure
auto result = make_lc_alpha("invalid123")
    >>= stemmer
    >>= make_ngram<2>;  // Returns nullopt
```

Listing 2: Monadic composition in Alga

3 Design Philosophy

3.1 Uniform Interface Through Concepts

Alga leverages C++20 concepts to enforce algebraic structure requirements at compile time:

```cpp
template<typename T>
concept AlgebraicType = requires(T const& a, T const& b) {
    { a * b } -> std::convertible_to<T>;
    { T{} };  // Identity element
};
```

Listing 3: Algebraic type concept

This ensures all parser types support the same algebraic operations, enabling generic algorithms that work with any conforming type.

3.2 Operator Algebra

Beyond basic concatenation, Alga implements a rich set of algebraic operators that follow mathematical laws:

- `*` : Concatenation (monoid operation)
- `|` : Choice (first successful parse)
- `^` : Repetition (Kleene star variant)
- `>>` : Sequential composition
- `&&, ||` : Logical combinations

These operators combine to form a complete algebra for parser composition:

Theorem 1 (Distributivity)**.** For parsers $p, q, r \in \mathcal{P}$: $p * (q|r) = (p * q)|(p * r)$

4 Implementation Techniques

4.1 Perfect Value Semantics

All Alga types implement complete value semantics, enabling use in standard containers and algorithms:

```
std::vector<lc_alpha> words;
words.push_back(*make_lc_alpha("hello"));

// Full move semantics support
auto word = std::move(words[0]);

// Container storage without restrictions
std::map<lc_alpha, porter2_stem> stem_cache;
```

Listing 4: Value semantics example

4.2 Template Metaprogramming

Template specialization ensures zero-cost abstractions while maintaining type safety:

```
template<size_t N, typename T>
class ngram_stem {
    std::array<T, N> elements;
public:
    // Compile-time size checking
    static_assert(N > 0, "N-gram␣size␣must␣be␣positive");

    // Type-safe composition
    auto operator*(ngram_stem const& other) const;
};
```

Listing 5: N-gram template specialization

5 Applications

Alga's design makes it particularly suitable for:

1. **Natural Language Processing**: The Porter2 stemmer and n-gram support provide building blocks for text analysis pipelines.
2. **Domain-Specific Languages**: The algebraic operator framework naturally extends to DSL implementation.
3. **Text Validation**: The Optional pattern provides elegant handling of validation failures without exceptions.

Example: Building a text processing pipeline:

```
auto process_text(std::string_view input) {
    return make_lc_alpha(input)
        >>= [](auto word) {
            return porter2_stemmer{}(word);
        }
        >>= [](auto stem) {
            return make_bigram(stem, stem);
        };
}
```

Listing 6: Practical text processing pipeline

6 Conclusion

Alga demonstrates that rigorous mathematical foundations need not compromise practical efficiency. By modeling parsers as algebraic structures and leveraging modern C++ features, we achieve a library that is both theoretically sound and practically useful. The uniform Optional monad pattern provides consistent error handling, while template metaprogramming ensures type safety without runtime overhead.

Future work includes extending the algebraic framework to support parallel composition and investigating category-theoretic generalizations of the parser algebra.

Acknowledgments

The design of Alga was influenced by the rich tradition of parser combinators in functional programming, particularly the work on monadic parser combinators by Hutton and Meijer, and the algebraic approach to language theory.