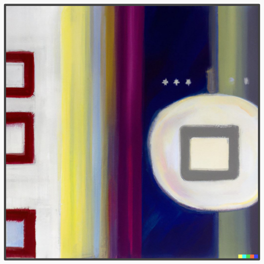
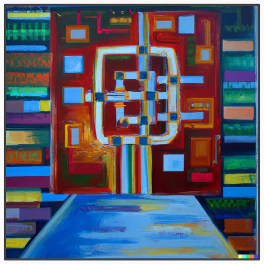
Applications of Algebra to Software Engineering

Chris Smith
Software Engineer, Groq, Inc.

Why Algebra in Software Engineering?

API Design   Composability

Abstraction   Correctness

Coming up: Loads of examples!

1

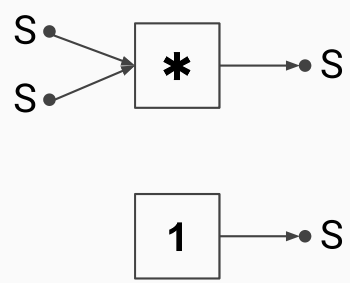
Designing with algebraic structure

What is an algebraic structure?

Types

S

Operations



Axioms

$$a * (b * c) = (a * b) * c$$
$$1 * a = a$$
$$a * 1 = a$$

Example: Regular expressions

Types

- Alphabet (A)
- Expressions (E)

Operations

- nothing : E
- empty : E
- literal : A → E
- concatenation : E × E → E
- alternation : E × E → E
- Kleene star : E → E

Axioms

- Associativities
- Identities
- Commutativity of alt.
- Idempotence of alt.
- Distributivity
- Annihilation
- Inclusion axioms for Kleene star

Examples: Codd's Relational Algebra

Types

- Labeled n-ary Relations
- Primitive Types

Operations

- union, difference, product
- projection, selection
- rename
- natural join
- equijoin
- semijoin
- antijoin
- division

Axioms

- Idempotence of selection
- Selection distributes of difference, intersection, and union
- etc. (there are lots!)

Examples: Semigroup Compression

Types

- Alphabet: A
- Compression Tokens: T

Operations

- len : $T \to \mathbb{N}$
- solo : $A \to T$
- popHead : $T \to A \times List(T)$
- popTail : $T \to List(T) \times A$
- tryMerge : $T \times T \to Optional(T)$
- split : $T \times \mathbb{N} \to List(T) \times List(T)$

Axioms

- len(solo(x)) = 1
- popHead(solo(x)) = (x, [ ])
- popTail(solo(x)) = ([ ], x)
- When tryMerge succeeds:
  - It is associative
  - len(tryMerge(a, b)) = len(a) + len(b)
- More axioms for split…

Examples: Mock Tests

Types: `Plan, Call`

```
empty : Plan
call : Call -> Plan
⧺ : Plan × Plan -> Plan
⫽ : Plan × Plan -> Plan
+ : Plan × Plan -> Plan

consec : Plan -> Plan
consec(p) = empty + p ⧺ consec(p)

multi : Plan -> Plan
multi(p) = empty + p ⫽ multi(p)
```

See Svenningsson J., Svensson H., Smallbone N., Arts T., Norell U., Hughes J. (2014) *An Expressive Semantics of Mocking*. In: Gnesi S., Rensink A. (eds) Fundamental Approaches to Software Engineering. FASE 2014. Lecture Notes in Computer Science, vol 8411. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-54804-8_27

Equational Theories: Pictures

Type: Picture

```
solidCircle : ℝ → Picture
solidRectangle : ℝ × ℝ → Picture
translated : Picture × ℝ × ℝ → Picture

◇ : Picture × Picture → Picture
```

$$\mathtt{trans}(a \diamond b, x, y) = \mathtt{trans}(a, x, y) \diamond \mathtt{trans}(b, x, y)$$

$$\mathtt{trans}(\mathtt{trans}(a, x_1, y_1), x_2, y_2) = \mathtt{trans}(a, x_1 + x_2, y_1 + y_2)$$

Live Demo

Benefits of Equational Theories

- Promotes confident refactoring
- Enables property testing
- Enables developer tooling
- Enables abstraction

- Encourages high-quality API design.
  - Operations are total and closed.
  - Properties like associativity, distributivity, idempotence, identities tend to appear.
  - Homomorphisms occur naturally.

APIs are algebraic structures.

(but we often neglect the axioms)

Concrete implementations are instances of those algebraic structures.

2

Programming with standard algebraic structures

Standard algebraic structures

Sometimes, the algebraic structures we need are already well-understood:

- Monoids and semigroups are all over the place!
- Semirings and lattices are also common.
- Several earlier examples exhibited monoid/semigroup structure:
  - Regular expressions are monoids under concatenation and alternation.
  - Compression was a partial semigroup under tryMerge.
  - Pictures are monoids under ◇.
- Groups, rings, and other structures with inverses are less common.

Abstracting over a structure

- Choose your own implementation of the structure!
- Goes by different names in different languages:
  - Bounded Type Parameters (Java)
  - Concepts (C++)
  - Type Classes (Haskell, PureScript)
  - Traits (Scala, Rust)
- Allows for very powerful abstraction boundaries.

Monoids

Abstractly: a set with an associative binary operation having an identity.

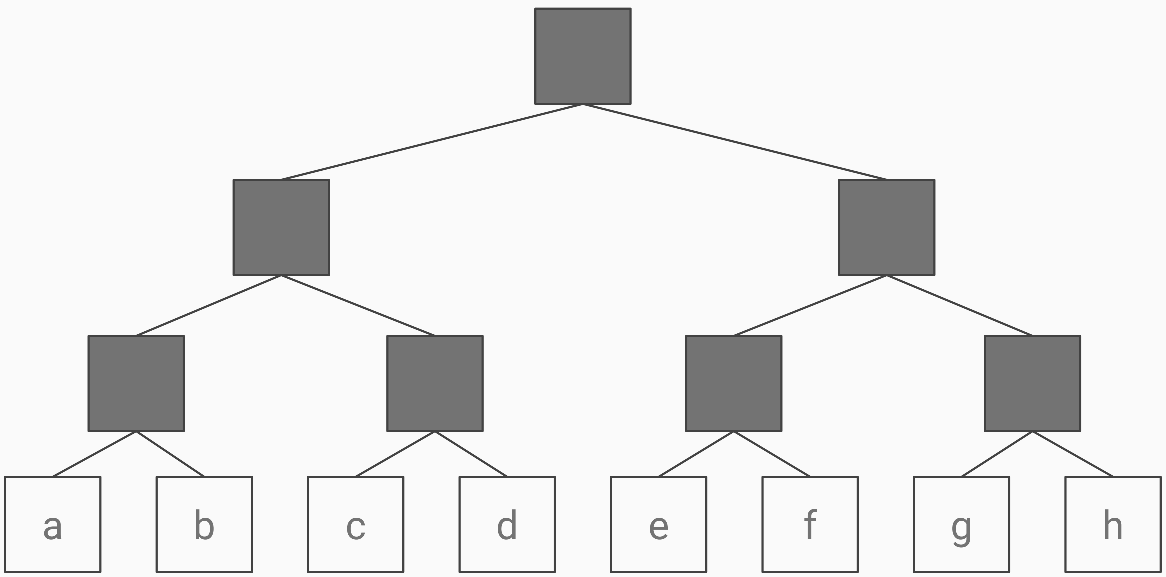
Concretely: a summary for lists that splits over list concatenation.

- Examples: sum, count, minimum, maximum, first, last, gcd, etc.
- Non-examples: mean, median

Many applications:

- Parallel and distributed algorithms
- Streaming and incremental computation

Balanced Trees



Sequences

Priority Queues

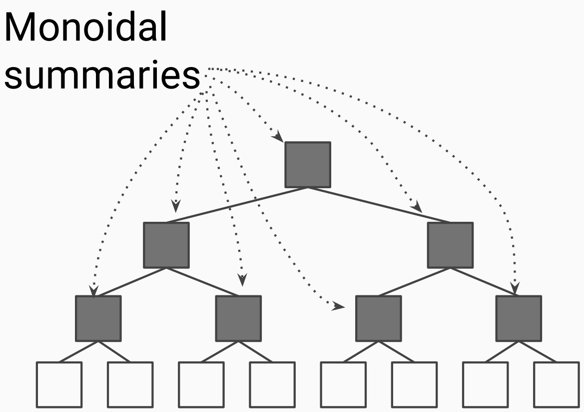
Interval Sets

Maps and Sets

Range Queries

Efficient insert, delete, split, concatenate, lookup, etc.

The Trick: Cache a summary in each node



Sequences: Cache the number of elements

Priority Queue: Cache the max element

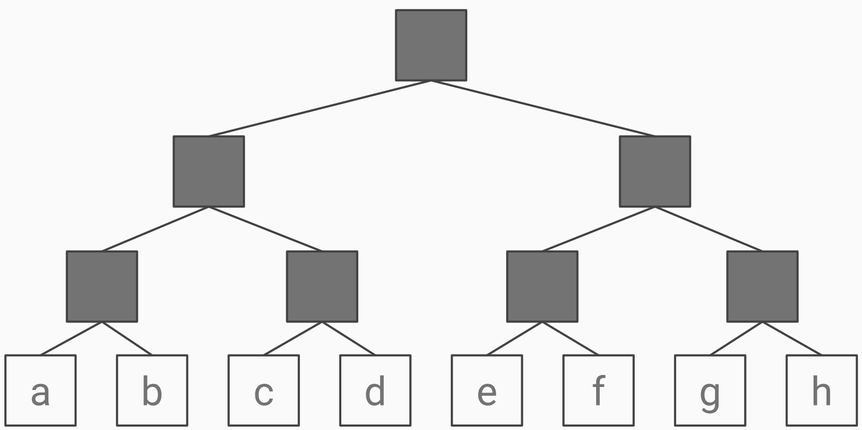
Interval Sets: Cache the smallest containing element

Maps and Sets: Cache the max key

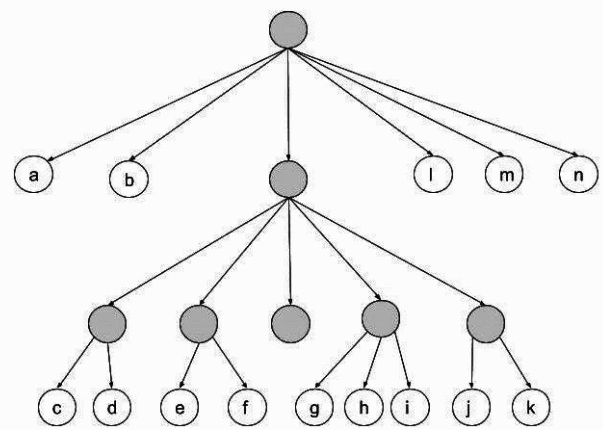
Range Queries: Cache the min and max elements

Associativity guarantees that rebalancing a subtree doesn't change its summary.

Swappable tree structure



Binary tree



2-3 Finger Tree

Implementations

- fj.data.fingertrees.FingerTree (Java)
- ttftree.Tree (Python)
- fingertrees::FingerTree (Rust)
- Data.FingerTree (Haskell)
- data.finger-tree (Clojure)

Algebraic structure can be abstracted over.

When the structure is very general, one can define very powerful APIs in this way.

3

Algebraic data types and the semiring of types

Operations on Sets/Types

Sum

$A + B = \{x_1 \mid x \in A\} \cup \{x_2 \mid x \in B\}$

Also known as:

- Disjoint union
- Tagged unions
- std::variant
- Enums in Scala 3, Rust, Swift, and more
- Subclasses

Product

$A \times B = \{(x, y) \mid x \in A, y \in B\}$

Also known as:

- Tuples
- Pairs
- Records
- Structs

Exponent

$A^B = \{f \mid f : B \to A\}$

Also known as:

- Functions
- Total Maps

The semiring of types

$A + B \cong B + A$
$(A + B) + C \cong A + (B + C)$
$A + 0 \cong 0 + A \cong A$

$A \times B \cong B \times A$
$(A \times B) \times C \cong A \times (B \times C)$
$A \times 1 \cong 1 \times A \cong A$

$A \times (B + C) \cong A \times B + A \times C$

$(A^B)^C \cong A^{B \times C}$
$A^{B + C} \cong A^B \times A^C$
$A^1 \cong A$

Types in a programming language form a semiring with exponentiation…

- Up to type isomorphism…
- In the category of *types* and *computable functions.*

Type variables and parametricity

- A data structure can be defined as a type parameterized by one or more simpler types.
- Often called generics or templates when implemented in a language.

- In this case, a function being *computable* carries a lot of weight.
- Parameter types must be handled in a formulaic way.
- This restriction is useful: it’s known as the parametricity theorem!

Metaprogramming and the semiring

- *Algebraic* types are definable with only products, sums, and fixpoints.
- Key result: by defining what should be done with products and sums, we can define a function on arbitrary algebraic types.
- (Optionally, handle exponents as well.)
- Metaprogramming facilities depend on programming language.

Example: Enumerating values of a type

$\text{enumerate}(A + B) = \text{enumerate}(A) \cup \text{enumerate}(B)$

$\text{enumerate}(A \times B) = \{ (x, y) \mid x \in \text{enumerate}(A), y \in \text{enumerate}(B) \}$

Example: Poking Holes in Data Structures

If T is a type, let $D_X(T)$ be the type of data structures with one missing value of type X.

- $D_X(C) = 0$ (when X doesn't occur in C)
- $D_X(X) = 1$
- $D_X(U + V) = D_X(U) + D_X(V)$
- $D_X(U \times V) = U \times D_X(V) + D_X(U) \times V$
- $D_X(F(G(X)) = D_{G(X)}F(G(X)) \times D_XG(X)$

Example: Tries

A trie $T(K, V)$ is an efficient map from K to V whose structure is dictated by K.

- $T(1, V) = V + 1$
- $T(A + B, V) = T(A, V) \times T(B, V)$
- $T(A \times B, V) = T(A, T(B, V))$

Types have algebraic structure.

We can exploit this structure for metaprogramming.

4

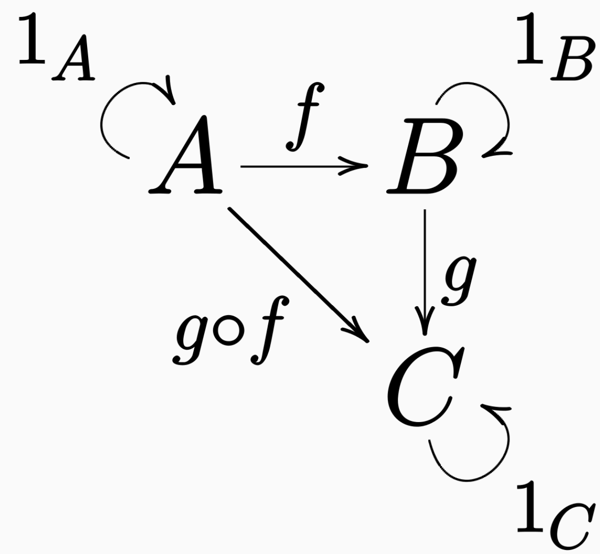
Categorical structure in software

Categories

- Objects
- Morphisms
- Identities
- Associative Composition

Category of Types

- Types
- Computable functions
- f(x) = x
- (g ∘ f)(x) = g(f(x))



Functors

- $f : A \to B$
- $map(f) : T(A) \to T(B)$

- Examples:
    - Lists or other data structures
    - Optional values
    - Placeholders for future values (promises, lazy computations, etc.)

Monoidal / Applicative Functors

- $f : A \times B \to C$
- $map(f) : T(A \times B)$ -> $T(C)$
- $map_2(f) : T(A) \times T(B) \to T(C)$

- Monoidal form: zip : $T(A) \times T(B) \to T(A \times B)$
- Applicative form: ap : $T(C^B) \to T(C)^{T(B)}$
  - Recall: $C^{A \times B} \cong (C^B)^A$

- Most commonly used functors are applicative, but not always uniquely.

Monads and Kleisli Categories

- $f : A \to_T B$ aka, $A \to T(B)$
- $g : B \to_T C$ aka, $B \to T(C)$
- $g \circ_T f : A \to_T C$ aka, $A \to T(C)$
- $id_{X,T} : X \to_T X$ aka, $X \to T(X)$

- Theorem: T is a monad, and Kleisli categories are in 1-to-1 correspondence with monads.

- Applications: too many to type!

More Functor Structures

Traversable Functors

- Assume F is another applicative functor.

- traverse : $F(B)^A \to F(T(B))^{T(A)}$
- sequence : $T(F(A)) \to F(T(A))$

- Examples: Any data structure

Alternative Functors

- T is an applicative functor. Additionally:

- empty : $T(A)$
- $\curlyvee : T(A) \times T(A) \to T(A)$

- Examples: Optional, Parsers, anything nondeterminism or recoverable failure

Example: Interpreters via Free Structures

Given a functor F (algebraic data type with a type param) that defines desired operations:

- Add : $A \times A \longrightarrow A$
- Negate : $A \longrightarrow A$
- Scale : $\mathbb{R} \times A \longrightarrow A$

The free monad generated by F defines a monad structure:

- Expr = Free(F)

If F is a functor, an F-algebra maps $F(X) \longrightarrow X$.

eval : $F(\mathbb{R}) \longrightarrow \mathbb{R}$
eval(Add(x, y)) = x + y
eval(Negate(x)) = -x
eval(Scale(k, x)) = x

An F-algebra defines an interpreter for the free monad.

interpret(eval) : $\mathbb{R}^A \longrightarrow \mathbb{R}^{Expr(A)}$

Example: Compiling to Categories

There is a canonical embedding of lambda calculus in any Cartesian Closed Category.

Idea: (Conal Elliot)

- Compile a programming language to categorical expressions
- Abstracted over the choice of category!
- Choose a category that works for the desired application.

Examples:

- Extract data flow graphs from programs.
- Convert rich programming languages to custom hardware.
- Automatic differentiation
- Incremental evaluation
- Reasoning about programs with constraint solvers (e.g., SMT)

Categories abstract function-like ideas.

Standard categorical structures are often applicable to programming problems.

5

Questions / Discussion