Information Algebra

by Sven Nilsen
version 002

0 – Motivation

It is hard to reason mathematically about data structures and algorithms that ignores the details of how it is implemented. How do we find which approach is best without writing a single line of code? What can we prove about specific computer programs?

Algebra is a practical and natural notion. We can use algebra to encode some knowledge about the problem we have and transform it into alternative approaches. By comparing different approaches, we can select the one that most probably will give us the greatest benefit. To do this we need to formalize the notion and explain what goes in the thinking process when reading the symbols. The notion should …

- … encourage reusable design when possible.
- … provide an easy overview over the 'kind' of data structure.
- … be programming language agnostic.
- … readable in source comments.
- … tested enough to be proven useful before publishing.
- … be compatible with Category Theory.
- … use modern conventions in programming.

1 - System

In this algebra, we use the word "system" as a word for anything that encodes information. A system is an abstract object that is composed with other systems. The sentence “X contains Y that contains Z” is written the following way, reading from left to right:

$$X \circ Y \circ Z$$

A “monoid system” is a system that encodes subsystems of same type, for example a tree of natural numbers:

$$\mathbb{N} \circ [\,] \circ \mathbb{N}$$

Example	**Description**	**Comments**
1	Unknown system	Has the capability of addressing an unknown system.
B	Binary system	Neighbors are bit.
$\mathbb{N}$	Natural number system	Neighbors differ by 1.
$[\,]$	Linear system	Used for lists and sets such as hash tables or arrays.
f, g	Functions, values	Starts with lower case.
(X, Y)	Tuple	Stores X and Y together.
$(X + Y)$	Union	Stores either X or Y but not both at same time.

2 - Data Structures

A composition of systems gives a data structure. Here is a list of natural numbers, encoded into a binary system:

$$B \circ [\,] \circ \mathbb{N}$$

When we are dealing with computer memory, there is always a 'B' at the top. If we have a union of X and Y in a binary system we can transform it into a union of the binary representation of X and Y:

$$B \circ (X + Y) \rightarrow (B \circ X + B \circ Y)$$

This process is called “flattening”.

Fundamental Theorem of Data Structures
$Z = Z \circ 1 = Z \circ (X + \neg X)$ $Z \neq (Z \circ X + Z \circ \neg X)$ In the flattening process there is loss of information.

The fundamental theorem is what makes the algebra useful, because we can transform one solution into another non-equal solution with different properties:

$$\mathbb{N} \circ (X + Y) \rightarrow \mathbb{N} \circ X + \mathbb{N} \circ Y$$
$$\mathbb{N} \circ (X + Y) \neq \mathbb{N} \circ X + \mathbb{N} \circ Y$$

3 - Addressing Capacity

For addressing capacity we use the following notation:

$|X|$

Example	**Description**	**Comments**
$\|1\|=1$	Unknown system capacity	Addresses just one system.
$\|B\circ \mathbb{N}\|=2^{\|B\|}$	Natural number capacity	The addressing capacity of natural numbers grows exponentially with the number of bits.
$\|B\circ[\,]\circ X\|=k\|B\|\|B\circ X\|$	Linear capacity	The capacity of a linear data structure grows linearly with the number of bits.
$\|X\|=\|1-\neg X\|$		The capacity of X is equal to the capacity of storing anything not X.
$\|(X,Y)\|=\|X\|\|Y\|$	Addressing capacity	The tuple of two systems can address the equal amount of their product.
$\|B^x\|=x$	Binary addressing capacity	The only addressing capacity we get from a binary system without peeking inside is the number of bits.

4 – Linear Operator

A linked list and an array both can do the same job. The linked list needs a pointer to the next element, which require more memory than the array. The array needs a number telling how large the array is. Still, they both grow linearly when expanding the memory with new bits.

$$|B \circ [\,] \circ X| = k|B||B \circ X|$$

When the constant is close to each or the upper bound is lower than the available memory, the implementations are consider equal. We use just one symbol [] that represents list, array, set, hash table, generator or any type that scales linearly.

For example, a map type can be represented as a list of tuples containing both the key and value:

$$[\,] \circ (key, value)$$

5 - Equality

Example	Description	Comments
$X \neq Y \Leftrightarrow X \in \neg Y \ , Y \in \neg X$	Different systems are exclusive	When we say "not X" this includes all systems that are not equal to X.
$X + \neg X = 1$	Law of unknown system	An unknown system is a choice from the union of any system and its complement.
$X \circ Y \neq Y \circ X$	Non-commutativity	A data structure where Y is encoded in X is not equal to the data structure where X is encoded in Y.

6 - Hypercube

Assume we have a graph where each node is represented with bits. The neighbors differ in bits by one, such that XOR operation between their value tells which bit is changing:

$$\begin{aligned} a &= 10101101101 \\ b &= 10101001101 \\ xor(a,b) &= 00000100000 \end{aligned}$$

This forms a binary lattice in hyperspace, a hypercube. We can separate all edges into exclusive groups based on their XOR operation. Each edge is uniquely defined as a member of one such group:

$$|\mathbb{N}_{edge}| = |(\mathbb{N}_g, \mathbb{N}_m)| = |\mathbb{N}_g||\mathbb{N}_m| = |B^n_{node}| \cdot 2^{|B^n_{node}|-1} = n \cdot 2^{n-1}$$

7 - Prime Number Monoid System

Prime numbers can be used to store lists in a single natural number. Every natural number, except 0, can be written as a product of prime numbers, where the exponent allows encoding of information that is not mixed with the other exponents:

$$\mathbb{N} \circ P \circ [\,] \circ X = 2^{x_0} \cdot 3^{x_1} \cdot 5^{x_2} \ldots$$

To read back this information, it requires factorization of the number. If we had infinite binary memory, we could construct this data structure:

$$B \circ (\mathbb{N} \circ P \circ [\,])^n \circ B$$

For each power of 'n', we could replace the current data structure with a new one that gives us infinitely more memory, but we must give up the data structure we have to replace it with a new one. This is because different values of 'n' results in different systems.

8 - Factoring Out Tuples

We have a tuple describing the a particle type and its mass:

$$(\mathbb{N}_p, \mathbb{R}_m)$$

A list of such tuples can be transformed into lists per type of particle. This process is called “factoring”. We say we are “factoring out the particle type”:

$$[]\circ(\mathbb{N}_p, \mathbb{R}_m) \rightarrow []\circ\mathbb{N}_p\circ[]\circ\mathbb{R}_m$$

Why did we not factor out the mass? Because the addressing capacity of the particle information is much lower. If we factored out the mass, it would require many more lists.

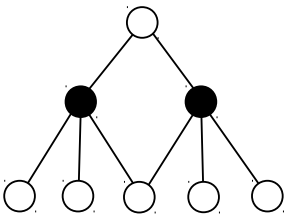
Factoring Rule
$[]\circ(X,Y) \rightarrow min(\lvert X\rvert,\lvert Y\rvert)\circ[]\circ max(\lvert X\rvert,\lvert Y\rvert)$

9 – Dependency Graph

A directed graph allows every node to point to any other node, including itself. The 'G' stands for 'graph' and 'D' stands for direction. The order in the tuple specify direction:

$$G \circ D = ([\,] \circ \text{Node}, [\,] \circ (\mathbb{N}_{\text{node}}, \mathbb{N}_{\text{node}}) \circ \text{Edge})$$

One kind of directed graph is useful for modeling dependencies. In the picture below, the “black” nodes require all of its dependencies to be fulfilled, while the “white” nodes require only one of the dependencies to be fulfilled. In programming the “black” nodes corresponds to algorithms and “white” nodes corresponds to types.



$$G \circ D_{w,b} = ([\,] \circ w, [\,] \circ b, [\,] \circ (\mathbb{N}_w, \mathbb{N}_b) \circ \text{In}, [\,] \circ (\mathbb{N}_b, \mathbb{N}_w) \circ \text{Out})$$