Information Algebra

by Sven Nilsen
version 008

Preface

Programmers today suffer from not seeing the great picture before they start to code. After 15 years of programming, I needed a way to push my intuition about programs a step further. Through an iterative process I came up with this notion as a result. It does not mean it is complete, neither is it a replacement for other powerful mathematical tools. It is designed to tickle my brain in a certain way. I call it an "information algebra" because it is meant to be used in equations and it is about the amount of information, not algorithms.

It is extremely hard to come up with something new in this field, so I expect people seeing the same ideas repeated elsewhere. I have not used any other source of inspiration, but have read about other things, such as category theory and abstract algebra. Because the language in these fields can be intimidating, I made a conscious effort to restrict the terms to a minimum set familiar for most programmers.

Abstract mathematics can also be ambiguous work. In the beginning there is no axioms, just a loose sense of what you are looking for. As you guess the rules of game, it starts bouncing back on itself and you can go back refining it. This leads to new ideas, new rules and the snowball keeps rolling and growing. What I set out to do, was understanding how we can build abstractions from compositions. The result was unexpected.

The algebra defines a set of rules that tells something about data structures even when we do not know how these data structures are implemented.

When developing this algebraic notion it required several iterations and lot of confusions of what it meant. It started out as a rigid idea about usage of computer memory and how data structures expanded in capacity when the number of bits increased. This idea worked for simple examples but became very soon too complex. It was replaced by a less strict convention where two data structures are similar if they approximately can be used to solve the same problem.

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 without having the source code?

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 these, we can select the one that gives the greatest benefit. To do this we need to formalize the notion and explain what goes in the thinking process when working with 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.
- … use modern conventions in programming.

The notion should strike a nice balance between abstracting over implementations while at the same time be very specific about the stuff that matters.

1 - System

We use the word "system" as a word for anything that encodes information. The sentence "X contains Y that contains Z" is written the following way, reading from left to right:

$$X \circ Y \circ Z$$

In plain text you can write it as "X*Y*Z". This is called a "composition" or "data structure".

For example, a linear system, such as a list or hash table, encoding a system X:

$$[\,]\circ X=[X]$$

The first way of writing "[]*X" is preferred because it gives better readability by not having to count the brackets.

Example	Description	Comments
1	Unknown system	Has the capability of addressing an unknown system.
$()$	Null system	Empty union or tuple is a null system, it encodes nothing.
?	Optional system	Encodes something or nothing.
B	Binary system	Neighbors are bit.
$\mathbb{N}$	Natural number system	Neighbors differ by 1.
$\mathbb{R}$	Rational number system	Usually implemented as floating point type.
$[\,]$	Linear system	Used for lists and sets such as hash tables or arrays.
(X,Y)	Tuple	Stores X and Y together.
$(X+Y)$	Union	Stores either X or Y but not both at same time.
f,g	Functions, values	Starts with lower case.

2 - Flattening

When we have a union of some systems encoded into another system, we can "flatten" the data structure. This process is analogous to multiplication in linear algebra. Flattening reduce memory usage and gives better type safety.

When a union of X and Y is encoded in binary, so it should be possible to encode X and Y independently in binary and use a union of the binary blobs. In practice we always want to be able to encode types independently to binary because it gives us more flexible choices of how to store data:

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

We use a different rule for linear systems. Instead of getting a new union we get a tuple of lists. This changes how we address each item. One the left side we can refer to each item using a single number. On the right side we need a union to distinguish which list we are referring to:

$$[\,] \circ (X + Y) : \mathbb{N}_{(X+Y)} = ([\,] \circ X, [\,] \circ Y) : (\mathbb{N}_X + \mathbb{N}_Y)$$

For example, when a linear system contains references to itself, these references must be transformed as well:

$$A \circ X = [\,] \circ (X + \mathbb{N}_A) = ([\,] \circ X, [\,] \circ (\mathbb{N}_{A0} + \mathbb{N}_{A1}))$$

Flattening Rule
For a non-linear type flattening results in a union: $$B \circ (X + Y) = (B \circ X, B \circ Y)$$ For a linear type flattening results in a tuple, with a transform of addressing: $$[\,] \circ (X + Y) : \mathbb{N}_{(X+Y)} = ([\,] \circ X, [\,] \circ Y) : (\mathbb{N}_X + \mathbb{N}_Y)$$

3 - Addressing Capacity

For addressing capacity we use the following notation:

$$|X|$$

The axioms about addressing capacity are very unlike each other, but work well together and becomes a powerful tool for analyzing data structures and algorithms.

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.
$|B \circ [\,] \circ X|=k_{B \circ X}|B||B \circ X|$	Linear capacity	The capacity of a linear data structure grows linearly with the number of bits.
$|B \circ (X,Y)|=|B \circ X||B \circ Y|$	Tuple 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 System

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.

$$|B \circ [\,] \circ X| = k_{B \circ X} |B| |B \circ X|$$

When the constant is close to each other or the upper bound is much lower than the available memory, when they have no noticeable difference in performance, the implementations are consider equal. Another way to think of it is given the description of the data structure and the spec, the programmer should be smart enough to choose the optimal implementation. We use just one symbol [] that represents list, array, set, hash table, generator for finite groups, monetized data or any type that scales linearly in optimal form.

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

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

The programmer should be smart enough to see what kind of data structure should go where. In the future an automatized analyzer might understand this algebra good enough be able to pick the optimal implementation by looking at a formalized spec format, maybe even transform the data structure to a better one.

As we derived earlier we can transform a list of unions into a tuple of lists:

$$[\,] \circ (X + Y) = ([\,] \circ X, [\,] \circ Y)$$

Another process we will take a look at later is "factoring", which can reduce memory usage:

$$[\,] \circ (X, Y) \rightarrow [\,] \circ min(|X|, |Y|) \circ [\,] \circ max(|X|, |Y|)$$

If a tuple is written on the left side of a linear operator, it means each system in the tuple encodes the stuff encoded into the list. Here is from an example we will discuss later called "alternate addressing":

$$(X, Y) \circ [\,] \circ \mathbb{N}_{(Y, X)}$$

5 – Tuple Capacity

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:

$$a = 10101101101$$
$$b = 10101001101$$
$$xor(a,b) = 00000100000$$

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:

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

We found the number of edges in a hypercube by using the law of tuple capacity:

Tuple Capacity Rule
$$|B \circ (X, Y)| = |B \circ X||B \circ Y|$$

6 – Self Hosting 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:

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

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. Because a such system is capable of replacing itself with a more advanced form, we call it "self hosting".

The self hosted prime number system is not recursive, because it does not address itself. It is easy to mistake self hosting systems for recursive and vice versa. An example is a graph data structure that can address itself:

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

This is not self hosting, but recursive. We can make it self hosting by encoding the whole graph as a number, but we need to distinguish between numbers as nodes and numbers as hosting:

$$\mathbb{N} \circ [] \circ (\mathbb{N}, [] \circ (\mathbb{N}_{node} + \mathbb{N}_{host}))$$

Now the data structure becomes self hosting in addition to recursive.

Self Hosting Rule
A self hosting system encodes the same type of system as it appears for the system it is encoded in. $\quad X \circ Y \circ X$

7 - Factoring

In a physics simulation, we have a tuple describing the a particle type and its mass:

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

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":

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

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
$$[\,]\circ(X,Y)\rightarrow[\,]\circ min(|X|,|Y|)\circ[\,]\circ max(|X|,|Y|)$$

8 – Tuple Addressing

Assume that we have a graph with edges, where the edges has 4 possible values:

$$|\text{Edge}| = 4$$

We define a tuple that gives us direction, that can be sorted if the graph is undirected:

$$D = \left(\mathbb{N}_{\text{from}}, \mathbb{N}_{\text{to}} \right)$$

We want to store direction consistently, which means we need to store it together with edge data. If we stored it per node then two nodes could disagree about the direction. We can factor out the value to save memory for edge data. In the first case we only need a number to address the edge. In the second case we need a tuple:

$$[\,]\circ (D, \text{Edge}) : \mathbb{N}_{\text{edge}}$$
$$[\,]\circ \text{Edge} \circ [\,]\circ D : \left(\mathbb{N}_{V}, \mathbb{N}_{D} \right)$$

We like to look up all edges connected to a node with O(1). This requires the address of the edge to be duplicated twice, one for each end. The tuple takes twice as much memory. To reduce memory we can create a function that maps from the first kind of address to a tuple:

$$f\left(\mathbb{N}_{\text{edge}} \right) \rightarrow \left(\mathbb{N}_{V}, \mathbb{N}_{D} \right)$$

The complete data structure looks as following:

$$\left([\,]\circ \left(\text{Node}, [\,]\circ \mathbb{N}_{\text{edge}} \right), [\,]\circ \text{Edge} \circ [\,]\circ \left(\mathbb{N}_{\text{from}}, \mathbb{N}_{\text{to}} \right) \right)$$

Tuple Addressing Rule
Tuple addressing means to encode multiple dimensions in the same number to save space. $$f\left(\mathbb{N}_{X} \right) \rightarrow \left(\mathbb{N}_{Y}, \mathbb{N}_{Z} \right)$$

9 – Alternate Addressing

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 equivalent algorithms and "white" nodes corresponds to types.
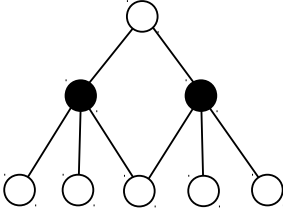
Assuming that we only need to know which nodes each node depends on, not which nodes depends on the node, we can use the two following data structures:

$$[\,]\circ(\text{White},[\,]\circ(\mathbb{N}_{\text{black}},\text{Edge}))$$
$$[\,]\circ(\text{Black},[\,]\circ(\mathbb{N}_{\text{white}},\text{Edge}))$$

This can be factored out and joined, by reversing the annotation of the address:

$$[\,]\circ(\text{White}+\text{Black})\circ[\,]\circ(\mathbb{N}_{(\text{black},\text{white})},\text{Edge})$$

Since we will be storing one list for white and one list for black, it falls naturally to write:

$$(\text{White},\text{Black})\circ[\,]\circ(\mathbb{N}_{(\text{black},\text{white})},\text{Edge})$$

A tuple on the left side of a linear system means we store the thing encoded in the list for each system in the tuple. This is called "alternate addressing".

Alternate Addressing Rule

$$(X,Y)\circ[\,]\circ\mathbb{N}_{(Y,X)}=((X,[\,]\circ\mathbb{N}_Y),(Y,[\,]\circ\mathbb{N}_X))$$

10 – Edge Cases

A union that does not contain 'X' but no other system is equal to a null system:

$$(-X)=()$$

The negation operator on a tuple with two elements can be used as a swap operator that satisfy Boolean algebra within a union. This can be used to create systems that store directional information:

$$Z=(X,Y)=\neg(Y,X)$$
$$(Z+\neg Z)\circ Z=Z$$

Some system encoded into a function can be interpreted as executing it with the system as argument:

$$f\circ X=f(X)$$
$$f\circ()=f(())=f()$$

A system 'Y' encoded into a system 'X' with no interpretation can be equal to the null system:

$$X\circ Y=()$$

11 – Inference

A person named Alice speaks Swedish and English:

$$A \circ L = (S + E)$$

Another person Bob speaks Danish and English:

$$B \circ L = (D + E)$$

Alice and Bob have a meeting.

$$(A \circ L, B \circ L) = (A, B) \circ L$$

$$(S + E) \wedge (D + E) = E$$

$$(A, B) \circ L = E$$

This is called "inference". When applying Boolean Algebra on unions we used the assumptions:

$$S \wedge D = ()$$
$$(S + D) \wedge E = ()$$

Inference Rule
$$(X \circ Z, Y \circ Z) = (X, Y) \circ Z = X \circ Z \wedge Y \circ Z$$

Boolean Algebra on Unions
$$(A + B) \wedge (C + D) = (A \wedge C + A \wedge D + B \wedge C + B \wedge D)$$ $$(A + B) \vee (C + D) = (A + B + C + D)$$

12 – Example: Trees

A tree structure can be modeled using lists of lists. We start out deriving the addressing capacity for a tree with only two levels:

$$|B \circ [\,] \circ X \circ [\,] \circ X|$$
$$|B \circ [\,] \circ (X, X)|$$
$$k_{B \circ (X, X)} |B| |B \circ (X, X)|$$
$$k_{(B \circ X, B \circ X)} |B| |(B \circ X, B \circ X)|$$
$$2 k_{B \circ X} |B| |B \circ X| |B \circ X|$$
$$2 k_{B \circ X} |B| |B \circ X|^2$$

Doing the same for three levels, four etc. we can generalize it:

$$|B \circ ([\,] \circ X)^n| = n k_{B \circ X} |B| |B \circ X|^n$$

For example, if we have 3 sub nodes per node the addressing capacity of the tree will grow with:

$$3 k |B| 3^n$$

What if the number of sub nodes are arbitrary? Assume that each node is assigned a natural number. The addressing capacity of a natural number is known, so we can plug it into the formula:

$$B \circ \mathbb{N} = 2^{|B|}$$
$$|B \circ ([\,] \circ \mathbb{N})^n| = n k_{B \circ \mathbb{N}} |B| 2^{|B| n}$$

Using natural numbers for addressing means the data structure can be recursive. In a deep data structure where there are lots of connections it is very likely that self reference occurs, because it quickly overflows the addressing capacity one uses to address each node:

$$2^{|B| n} > 2^{|B|}$$

Traversing a deep tree with lots of connections gives more information than the possible number of nodes. The traversing information is not restricted by the number of bits available but by computing capacity. When this information is greater than the bits used to address each node, it is guaranteed that the tree contains duplicates.

13 – Example: Family of Sets

We have a hierarchy of functionality where new features can be added to children that does not exist for the parents, or the children can override the behavior of specific functions. Assuming that we have access to one child node, we want to look up the right function in the context of that child. A sensible solution to this problem would be:

$$F \circ X = [\,] \circ ((\,(\,) + \mathbb{N}_{parent}), [\,] \circ \mathbb{N}_{child}, [\,] \circ (\mathbb{N}_{key}, X))$$

Using the optional system we can write this nicer:

$$F \circ X = [\,] \circ (? \circ \mathbb{N}_{parent}, [\,] \circ \mathbb{N}_{child}, [\,] \circ (\mathbb{N}_{key}, X))$$

The reason parent can be optional is that one node has to be the top node.

Family of sets can be used to organize a namespace.

14 – Addressing Duality

Assume we have the following data structure:

$$[\,]\circ X : \mathbb{N}_X$$

Each item in the data structure is addressable by an integer. For every data structure, there is an address data structure that encodes the information necessary to identify each item. When we transform the data structure, we also transform the information to address it. When information to address is encoded into a number, we can write the same data structure in a different way:

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

Here is another example:

$$[\,]\circ X \circ [\,]\circ Y : \left(\mathbb{N}\circ X + \left(\mathbb{N}\circ X, \mathbb{N}\circ Y\right)\right)$$

We can transform this data structure into an equivalent form:

$$[\,]\circ(X,Y) : \mathbb{N}\circ(X,Y)$$

If we would like to group the data by X, we can use the first form.

When we work with tuples, we can address the content with a union:

$$\left([\,]\circ X, [\,]\circ Y\right) : \left(\mathbb{N}\circ X + \mathbb{N}\circ Y\right)$$

Of course, when one has a tuple one can address multiple items simultaneously:

$$\left([\,]\circ X, [\,]\circ Y\right) : \left(\mathbb{N}\circ X, \mathbb{N}\circ Y\right)$$

The way we address information in a data structure is itself a data structure.