Information Algebra

by Sven Nilsen
version 004

Preface – Leap of Intuition

Alice has never heard of multiplication. One day she wakes up having a brilliant idea that there is an operator that makes it possible express repeated additions easier. She creates a table where she combines numbers with the new operator:

	1	2	3
1	1·1	1·2	1·3
2	2·1	2·2	2·3
3	3·1	3·2	3·3

The only thing Alice knows is that adding 1 one times is 1.

$$1 \cdot 1 = 1$$

Bob knows a secret rule that allows to reduce every problem into a sum of one by ones. Each time Alice need to multiply two numbers, she rewrites the numbers as sum of 1s, send them to Bob and gets the answer back:

$$(1+1) \cdot (1+1+1) = 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1$$

Does Alice knows multiplication? No, because she does not know the secret rule.
Does Bob knows multiplication? No, because he does not know how to add.

Does my brain knows multiplication?

When I stop and think about this for a moment, I realize that the brain does tricks similar to what Alice and Bobs do. When I use a calculator the brain encodes the number into finger movements and recalls memory about how to read the result from the screen. The whole process is vastly more complicated than what computers do. I can not understand multiplication in a single instant, but I am using a combination of systems to get the right answers.

In Information Algebra, there is no numbers or source code that makes it intuitive of how to interpret the result. It takes away the illusion of “knowing” and leaves me alone with the description of how it is put together. For example, when I read stuff like this:

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

The left side is less intuitive than the right side for me of how I would program it. My brain still struggles with making the leap of intuition that these two things are equivalent, across a numerous ways of implementing this in code.

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 before publishing.
- … use modern conventions in programming.

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.

Another idea that was abandoned was having “1” as the set of all systems in an imaginary physical sense. This turned out to be inconsistent with rest of the notion and changed to represent an “unknown” system. By coincidence the way the “union” is used it strictly means “union of all systems”, so it is the set of all systems in a logical sense.

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

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:

$$X \circ Y \circ Z$$

Notice it does not mean “sub structure” but “encoded into”. To create sub structures one uses tuples and unions.

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.
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

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”. When the union of 'X' and 'Y' can be encoded in binary, then it should be possible to encode them independently in binary. If we write encoding algorithms inspired by flattening process, we only need a general way of encoding unions to transform from binary form to a union.

Flattening can be applied to any system. Here we had a linear system of unions and flattened into a union of two lists:

$$[] \circ (X + Y) \rightarrow ([] \circ X + [] \circ Y)$$

If it is possible to list 'X' and 'Y' it should be possible to list them independently without the union information. Notice that this does not create two lists. It only says it is possible to have a list of each. This possibility can be used to derive a new equivalent data structure, a tuple of lists:

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

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

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

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:

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

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:

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

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:

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

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

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

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:

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

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. 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:

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

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

Self Hosting Rule
$X \circ Y \circ X$

7 - Factoring

In a physics simulation, 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
$[]\circ(X,Y) \rightarrow []\circ min(\lvert X\rvert,\lvert Y\rvert)\circ[]\circ max(\lvert X\rvert,\lvert Y\rvert)$

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:

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

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:

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

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:

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

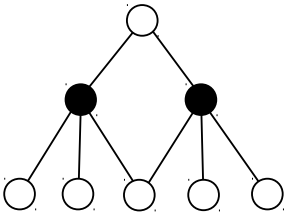
The complete data structure looks as following:

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

Tuple Addressing Rule
$f(\mathbb{N}_{X})\rightarrow(\mathbb{N}_{Y},\mathbb{N}_{Z})$

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 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:

$$\text{Alice} \circ \text{Language} = (\text{Swedish} + \text{English})$$

Another person Bob speaks Danish and English:

$$\text{Bob} \circ \text{Language} = (\text{Danish} + \text{English})$$

Alice and Bob have a meeting.

$$(\text{Alice} \circ \text{Language}, \text{Bob} \circ \text{Language}) \rightarrow (\text{Alice}, \text{Bob}) \circ \text{Language}$$

$$(\text{Swedish} + \text{English}) \wedge (\text{Danish} + \text{English}) = \text{English}$$

$$(\text{Alice}, \text{Bob}) \circ \text{Language} = \text{English}$$

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

$$\text{Swedish} \wedge \text{Danish} = ()$$
$$(\text{Swedish} + \text{Danish}) \wedge \text{English} = ()$$

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

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:

$$
\begin{aligned}
&|B \circ [\,] \circ X \circ [\,] \circ X| \\
&|B \circ [\,] \circ (X, X)| \\
&k|B||B \circ (X, X)| \\
&k|B||(B \circ X, B \circ X)| \\
&k|B||B \circ X||B \circ X| \\
&k|B||B \circ X|^2
\end{aligned}
$$

Doing the same for three levels, four etc. it is easy to see that:

$$|B \circ ([\,] \circ X)^n| = k|B||B \circ X|^n$$

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

$$k|B|3^n$$

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:

$$
\begin{aligned}
B \circ \mathbb{N} &= 2^{|B|} \\
|B \circ ([\,] \circ \mathbb{N})^n| &= k|B|2^{|B|n}
\end{aligned}
$$

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.