On aspects of Mathematical Methodology: Navigational Expressivity by way of Successor Spaces, Functional Compression

Daniel Nikpayuk

September 27, 2019

In the following essay I present independent research regarding the methodology of mathematics, focusing in particular on ideas of *navigation* within mathematically defined spaces. Before doing so, I need to introduce a little in the way of meta-mathematics and philosophy, along with a few prerequisite concepts that otherwise make up the remainder of this article.

methodology?

The word **methodology** comes from philosophy, the humanities, and the social sciences, but it makes sense to redefine it here so that we can agree on terms before getting started. For our purposes methodology is the logic of mathematical methods: If mathematicians study "mathematical things", then specifying a methodology is about saying *which* mathematical things we do study and *why*.

Looking at the language of math (disregarding alternative foundations) the things we generally talk about are *structures* and *functions* (which act on those structures). In some sense or another these are the nouns and verbs of this language. The question then becomes: Which structures? Which functions? Why do we value some functions more than others? Why do we *privilege* some structures over others? What makes them special?

If mathematical language is a means to an end, if mathematical objects are the *methods* to our madness, methodology aims to determine the ends to that madness.

entropy

A common observation in math is that there's a tradeoff between *utility* and *generality* when it comes to theorems. To put the idea casually: The more useful a theorem is, the less it says in general. The flipside is the more general the theorem the less it can actually do in particular.

For the remainder of this essay this theme will show up again and again, but I would prefer to reframe it in terms of *entropy*: When something has higher entropy it can at an intuitive level be thought to have less rigid structure, which is to say it's more flexible, adaptable (depending on context), or even random. On the other hand something with lower entropy is more rigid allowing for fewer possibilities, but the nature of the rigidity implies deeper patterns (and thus meaning) within the structure itself. In this sense highly general theorems are high entropy while low generality (thus high utility) theorems are low entropy—entropy of truth that is.

I claim then that it is a mathematician's methodology that mathematical "things" are valued when they have a *goldilocks* level of entropy: Not too much, but not too little either.

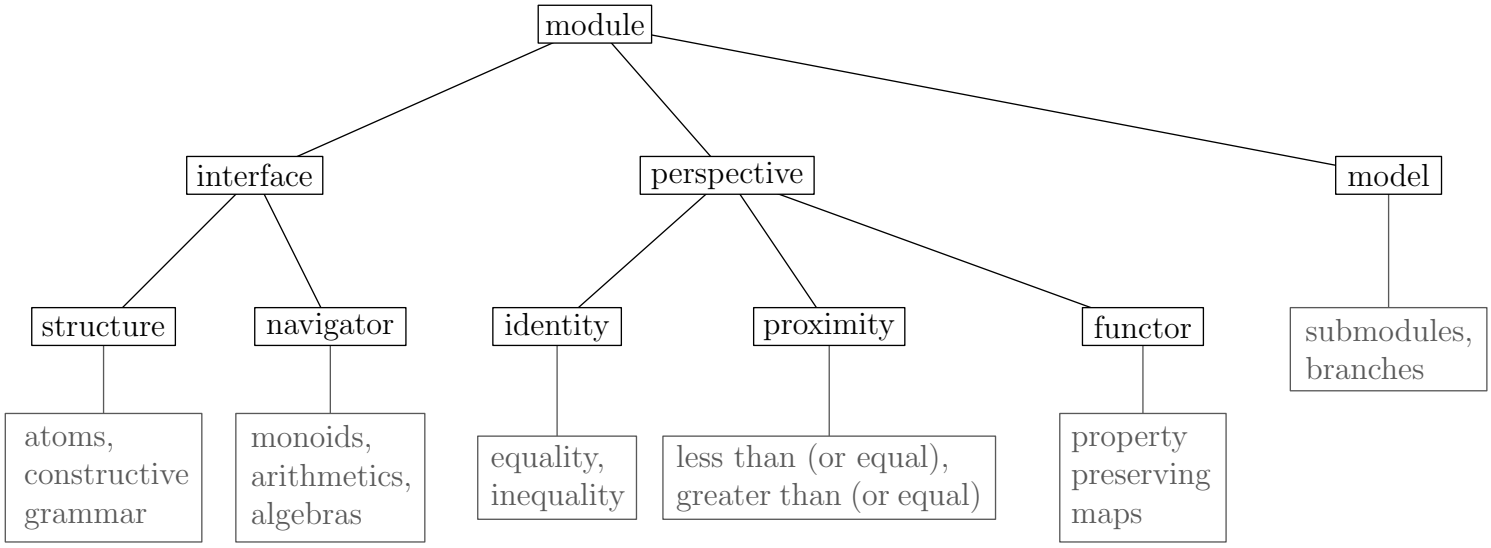
structural methodology

Generally speaking, the structures mathematicians study are organized as *spaces*, and it is my claim that of all the spaces chosen for study—the ones that are well researched and used in practice—are often the ones that allow us to interpret many other similar or simpler spaces as substructures. This methodology is called *modelling*, and can be considered a form of goldilocks entropy.

For example $\mathbb{R}^N$ ($N \in \mathbb{N}$) is well studied in Real Analysis, but $\mathbb{R}^3$ in particular is studied in greater detail because it both has a lot of structure—enough to say many meaningful things—and also allows us to model and interface with so many other (real world) structures, which is further applicable to physics, engineering, and so on.

I will further make the claim here that the spaces which make for ideal models are those spaces which are highly *entropic* when it comes to substructuring. You can get a sense of this by asking basic questions such as: Is it easy to express various substructures? How many substructures are there? How many kinds of substructures are there? I will add that highly substructurable objects aren't necessarily constrained to being self-similar, but in my experience such objects usually make for good candidates.

With this said I present here a "module" paradigm for methodologically valuable structures:



In the broadest sense here, this paradigm says that a structure would be good for modelling if you could otherwise partition and regroup its elements with ease into many different substructures. This larger process can be broken down as follows:

1. **interface**

   It's something we take for granted, but to partition and regroup the elements of a structure the first step is to interact with said structure, to interface with it.

   (a) **structure:** The design of any interface inevitably rests upon the way its underlying structure is defined and constructed. This is the usual place to start, especially as it informs the other designs.

   (b) **navigator:** Once we've determined the rules of construction, the usual next step is in figuring out what tools are available when it comes to navigating and accessing a structure's elements.

2. **perspective**

   Assuming an ability to pinpoint the elements of a structure, the next step is to compare them. As there is no single best measures of comparison in a general sense, the design of our module will offer various perspectives.

   (a) **identity:** The natural starting comparison is usually to determine whether two objects are equal or not.

   (b) **proximity:** If two objects aren't equal, the next natural step is to try to figure out just how similar or close they are. What direct comparisons are available?

   (c) **functor:** Finally, if two objects aren't equal, and there are no direct comparisons we are left with indirect methods. Are there ways to compare our elements to objects within an entirely different structure? Ways that might allow us to indirectly compare our elements of interest?

3. **model**

   Once we're freely able to compare the elements that make up a structure, we should then also be able express substructures readily. Now we can start the modelling process.

The last thing to mention about this module paradigm is that it is a *closed definition*. Its potential comes from the fact that it is also recursive: Once a model is achieved it can be reinterpreted as its own module which may in fact reuse the interface and perspective components from its superordinate module, or otherwise optimize its own.

functional methodology

Which functions and which types of functions do mathematicians prefer? And why? I make the claim that in the most general sense functions are considered valuable when they're considered *expressive*. Here I would define "expressivity" as a form of goldilocks entropy. Unfortunately this claim—this intuition, this value—is still too abstract so we will need to narrow it down, which we can rephrase by asking: Expressive? Expressive how?

I don't think there's any single best interpretation to this, but I have also realized after some thought that the structural methodology I have just presented also supplies an answer: Functional expressivities can be indexed against module divisions. So for example functions within the *proximity* division are considered expressive based on how well they let us make direct comparisons.

To this end, each division then has its own paradigms for functional expressivity. The breadth, depth, and variety of such paradigm divisions extend beyond the scope of this essay of course. Instead the focus here is on just one such divisional paradigm: Navigational expressivity. This leads us first to the idea of a *successor space*.

successor spaces

Given that this line of research is currently ongoing I'm reluctant to as of yet give a formal definition, but the overall idea is that a successor space is one which has an indexed family of "successor" functions:

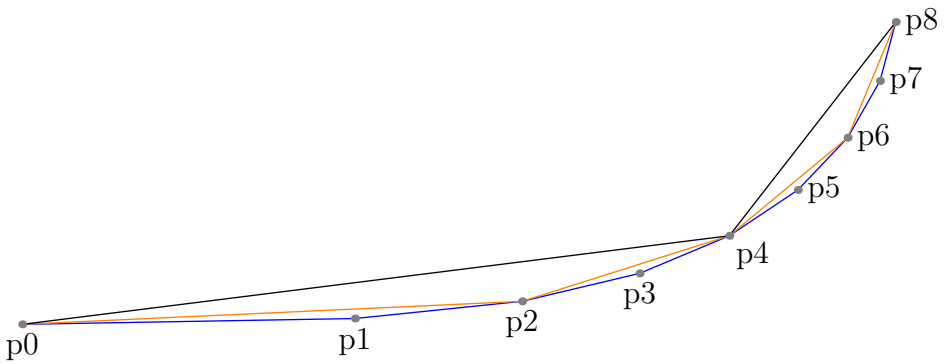
$$\{\mathrm{succ}_\alpha\}_{\alpha\in\mathcal{I}}$$

In particular, the idea is that *higher order* successors allow us to navigate across the space's elements at a lower cost, though I have yet to make precise this idea of "cost". Currently and maybe not unexpectedly I'm leaning toward *cost functions* (metrics) one typically sees in the sciences and engineering.

Taking a step back and looking at this from a philosophical lens, the idea here could be interpreted as a *vehicle*. Of course most recently "vehicles" are associated with motor vehicles, but the word itself has been around long before that, otherwise meant to express an idea of conveyance, or transportation. A vehicle then being a *tool* to move people or other things from one place to another.

The methodological ideal I'm looking to communicate here is that a vehicle and thus a successor space can be considered valuable since they represent important *energy saving strategies*. In life and in society humans are toolmakers, and vehicles (being one such tool) are often used to lower our own energy costs.

That being said I present the following graphic showing a sequence of points $p_0, p_1, \ldots, p_8$ as a visual aid:



These points are all connected directly to each other through paths (in blue), and if we're travelling with these points as our destinations, then these paths are our initial vehicles (walking could be considered such a vehicle). With that said, life looks to save energy when it can, and humans are no exception. If we're travelling these paths regularly we will in the long run probably want to seek out shorter paths, or vehicles that cost less.

In the above visual aid such energy saving vehicles would be the paths first in orange, then in black. This graphic can safely assume the triangle inequality, so if you're starting at p_0 and looking to arrive at point p_3 it's faster to "orange over" to p_2 then "blue it" one more step. The slower alternative of course would be to iterate only over blue paths to get there.

As for the terminology "successor space", the word "successor" comes from the successor function **succ** used in the definition of the natural numbers $\mathbb{N}$, which semi-formally would be:

$$n \in \mathbb{N} \quad \iff \quad n = 0 \quad \text{or} \quad \exists m \in \mathbb{N} \quad \text{such that} \quad n = \text{succ}(m)$$

If we apply the intuition from this definition, then a successor space should have an *initial element*, and all other elements are either its direct successor, or the successor of some already existing successor. If we extend this idea to higher order successors, and our cost function effectively says these successors are "skipping" elements as they enumerate, it is then implied that they enumerate over subsets of the full space. This is to say: Higher order successors are defined over subspaces.

The last intuition I wanted to mention before moving on is that our representative successor function **succ** may be simply defined, but more general successor functions will usually coincide with recursive functions. In fact several interesting examples require this.

example: radix notation

The best known example of a successor space is $\mathbb{N}$ itself:

$$\begin{aligned} \text{succ}_0(x) &= x + 10^0 \\ \text{succ}_1(x) &= x + 10^1 \\ \text{succ}_2(x) &= x + 10^2 \\ \text{succ}_3(x) &= x + 10^3 \\ &\vdots \end{aligned}$$

The initial successor function succ_0 corresponds to the defining successor function **succ**. In its potential this function is sufficent to navigate toward (and access) any given natural number, and yet mathematicians—in the few instances we're known to even use actual *number constants*—more often than not rely on higher order successors in the form of *decimal notation*, for example:

$$1024 = 1 \cdot 10^3 + 0 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0$$

which if we were to reframe this in terms of the successor space can be represented as the composite (vehicular) function:

$$1024 = \text{succ}_0 \circ \text{succ}_0 \circ \text{succ}_0 \circ \text{succ}_0 \circ \text{succ}_1 \circ \text{succ}_1 \circ \text{succ}_3(0)$$

Mind you this is the composite of *seven* successor functions, which may seem tedious but is still certainly better than *one-thousand-twenty-four* such initial functions.

example: real numbers

Probably the second best known example of a successor space is $\mathbb{R}$, this time using extended decimal notation:

$$\begin{aligned} &\vdots \\ \text{succ}_{-3}(x) &= x + 10^{-3} \\ \text{succ}_{-2}(x) &= x + 10^{-2} \\ \text{succ}_{-1}(x) &= x + 10^{-1} \\ \text{succ}_0(x) &= x + 10^0 \\ \text{succ}_1(x) &= x + 10^1 \\ \text{succ}_2(x) &= x + 10^2 \\ \text{succ}_3(x) &= x + 10^3 \\ &\vdots \end{aligned}$$

Here the index set for this family of successor functions ends up being the integers $\mathbb{Z}$. It should be said that finite compositions of these successors would not allow us to navigate all of the real numbers (actually only a subset the rationals $\mathbb{Q}$), so if this successor space were to be effective we'd have to accept countably infinite compositions of successor functions, or at least convergent sequences of finite compositions.

In any case, this example actually brings up an important insight which mathematicians might take for granted:

$$\lfloor x \rfloor \quad , \quad \lceil x \rceil$$

Respectively the *floor* and *ceiling* functions are—as it turns out—less a property of the real numbers and more a property of *higher order successors*. The insight then is if higher order successors exist, so then do floor and ceiling functions:

$$\lfloor x \rfloor_\alpha \quad , \quad \lceil x \rceil_\alpha$$

these of course being respective of their successor indices.

example: monotone spaces

Before moving on I wanted to give a non-obvious example of a successor space, to show that as a concept it might have the potential to assist in the study of mathematics more broadly.

To define this non-obvious space we need to start with integer valued tuples $\mathbf{u} \in \mathbb{Z}^k$, which is to say $\mathbf{u}$ is an object of the form $(u_1, \ldots, u_k)$, where $u_1, \ldots, u_k \in \mathbb{Z}$. Now let's restrict this space further to monotonically increasing tuples:

$$\mathcal{M}_k \quad := \quad \{\mathbf{u} \in \mathbb{Z}^k \mid u_1 \leq u_2 \leq \ldots \leq u_k\}$$

Finally, we can restrict this further to define *monotone* spaces:

$$\{m \leq \mathcal{M}_k \leq n\} \quad := \quad \{\mathbf{u} \in \mathcal{M}_k \mid m \leq u_1, \text{ and } u_k \leq n\}$$

As an aside, this might seem like a pretty artificial example, but these spaces show up a lot in sums of the form:

$$\sum_{0 \leq \mathbf{u} \leq n}^{k} f(\mathbf{u})$$

where the k above the sigma Σ indicates the dimension of the monotonic tuple $\mathbf{u}$. When $f(\mathbf{u}) := 1$ this sum counts the number of elements in its respective monotone space, which as it turns out ends up being:

$$\sum_{0 \leq \mathbf{u} \leq n}^{k} 1 = \binom{n+k}{k}$$

The merit of this style of summation is that it has several basic and important identities:

$$\sum_{0 \leq \mathbf{u} \leq n}^{k} f(\mathbf{u}) \quad = \quad \sum_{(0 \leq u_k \leq n)} \sum_{(0 \leq \mathbf{u} \leq u_k)}^{k-1} f(\mathbf{u}; u_k)$$

or

$$\sum_{0 \leq \mathbf{u} \leq n}^{k} f(\mathbf{u}) \quad = \quad \sum_{0 \leq \mathbf{u} \leq n}^{k-1} f(0; \mathbf{u}) \quad + \quad \sum_{1 \leq \mathbf{u} \leq n}^{k} f(\mathbf{u})$$

which are due to the self-similar nature of $\{m \leq \mathcal{M}_k \leq n\}$, but I digress.

Let's look now at a single example of this type of space $\{0 \leq \mathcal{M}_3 \leq 4\}$, it being our space of interest:

(0, 0, 0)	(1, 1, 1)	(2, 2, 2)	(3, 3, 3)	(4, 4, 4)
(0, 0, 1)	(1, 1, 2)	(2, 2, 3)	(3, 3, 4)	
(0, 0, 2)	(1, 1, 3)	(2, 2, 4)		
(0, 0, 3)	(1, 1, 4)			
(0, 0, 4)				
(0, 1, 1)	(1, 2, 2)	(2, 3, 3)	(3, 4, 4)	
(0, 1, 2)	(1, 2, 3)	(2, 3, 4)		
(0, 1, 3)	(1, 2, 4)			
(0, 1, 4)				
(0, 2, 2)	(1, 3, 3)	(2, 4, 4)		
(0, 2, 3)	(1, 3, 4)			
(0, 2, 4)				
(0, 3, 3)	(1, 4, 4)			
(0, 3, 4)				
(0, 4, 4)				

If you read the elements top-down, then left-to-right, you'll notice this particular iteration of them is *lexicographically* ordered. The tuples are color coded, with the intent of showing that the ones in orange are the first (higher) order successors, while the ones in black are the second.

I won't formally prove anything here, but the outline to defining this type of successor space more generally is as follows: We'd first acknowledge the zero tuple $\mathbf{0} := (0, \ldots, 0)$ as the initial element. After that we'd show these spaces' lexicographical orderings correspond with well-orderings which then allow us to define our primary successor succ_0 for these spaces, as well as the higher order ones.

We would define succ_1 as follows: First we would subset the monotone space:

$$\{m \leq \mathcal{M}_k \leq n\}_1 \quad := \quad \{\mathbf{u} \in \{m \leq \mathcal{M}_k \leq n\} \mid u_{k-1} = u_k\}$$

to which we would then derive the successor through this subset's sub-ordering.

This of course would generalize to the other higher order successors when they exist. For example the well-ordering of:

$$\{m \leq \mathcal{M}_k \leq n\}_2 \quad := \quad \{\mathbf{u} \in \{m \leq \mathcal{M}_k \leq n\} \mid u_{k-2} = u_{k-1} = u_k\}$$

would be used to define succ_2. In our above example $\{0 \leq \mathcal{M}_3 \leq 4\}$ it doesn't make sense for succ_3 and higher to exist in this space so we would stop here in this case.

The advantage of defining things this way is we can readily define the respective floor, ceiling, and max functions:

$$\begin{aligned}
\lfloor x \rfloor_j &:= \text{The greatest } \{m \leq \mathcal{M}_k \leq n\}_{j+1} \text{ less than or equal to } x \\
\lceil x \rceil_j &:= \text{The least } \{m \leq \mathcal{M}_k \leq n\}_{j+1} \text{ greater than or equal to } x \\
\max_j(x) &:= \text{The greatest } \{m \leq \mathcal{M}_k \leq n\}_j \text{ less than } \lceil x \rceil_j
\end{aligned}$$

Finally, if we wanted to define these successors in a more functional style we could implement them recursively going from higher to lower:

$$\begin{aligned}
\text{succ}_2 &: \quad \{0 \leq \mathcal{M}_3 \leq 4\}_2 \to \{0 \leq \mathcal{M}_3 \leq 4\} \\
\text{succ}_2(x) &:= \begin{cases} x + (1,1,1) & \text{if} \quad x \neq (4,4,4) \\ x & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\text{succ}_1 &: \quad \{0 \leq \mathcal{M}_3 \leq 4\}_1 \to \{0 \leq \mathcal{M}_3 \leq 4\} \\
\text{succ}_1(x) &:= \begin{cases} x + (0,1,1) & \text{if} \quad x \neq \max_1(x) \\ \lceil x \rceil_1 & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\text{succ}_0 &: \quad \{0 \leq \mathcal{M}_3 \leq 4\} \to \{0 \leq \mathcal{M}_3 \leq 4\} \\
\text{succ}_0(x) &:= \begin{cases} x + (0,0,1) & \text{if} \quad x \neq \max_0(x) \\ \lceil x \rceil_0 & \text{otherwise} \end{cases}
\end{aligned}$$

The general case can be defined similarly. Of course proofwise it would need to be verified that these definitions correspond with the well-ordering versions.

navigational expressivity

What makes a function navigationally expressive?

Referring back to successor spaces, the underlying methodology when it comes to acknowledging their navigational expressivity is that they are vehicles which allow us to navigate spaces at lower costs. The thing is, they are *a collection of functions* in which expressivity is emergent. What would make a single function navigationally expressive?

Reflecting on successor spaces as our inspiration, we might benefit from a rephrasing of this: A navigational function or path that lowers the cost of travel alternatively creates a surplus of energy; a surplus of potential; a surplus of entropy. In this sense, sometimes a function with greater expressivity is little more than a function with *further* expressivity. To that end, the methodology here can be summarized in a word: *compression*.

In practice, functions usually represent not just mappings to points in their given spaces, but *patterns* of such points. So if you're looking to navigate to a specific point you figure out a pattern it has and a corresponding function that knows how to arrive at that point by means of its pattern. Such a function is often a composite of simpler functions representing simpler patterns of course. Finally, there's often more than one way to arrive at a given point—not to mention more than one way to construct a given function—so of all the ways to get there we want a *minimal* path, and that's where compression comes in.

I will claim it is mathematical methodology that navigational functions are valued when they are *highly compressible*.

As a corollary, I would like to note that within this reasoning there is an additional necessary condition for expressivity: composability. For starters, a function $g : A \to S$ that acts *functorially*, relating A to our space S is moot as it implicitly shifts the weight of navigation to A. In this case we can instead keep things simple and restrict ourselves to functions $f : R \subseteq S \to S$ which when sufficiently composable allow us (as a necessary prerequisite) to build the functions used in our navigations.

What then makes a function highly compressible?

structural compression

We start by reviewing *structural compression*, which comes from information theory. The idea of compression in this theory is to figure out the distribution of words in a given text, then reencode those words such that the most frequent end up having the shortest encodings, thus compressing the structure of the text. Information theory even has its own specific way to measure entropy:

$$H(X) \quad = \quad -\sum_{x \in X} p(x) \log_2 p(x)$$

which gives a theoretical limit as to just how much a text can be compressed.

This is relevant to us because in practice navigating the functions of a space follows a well known *schema*: Define the primitive functions, then define composite functions as combinations of those primitives. If we want to compress the representation of a function a natural starter strategy is to look at its construction and figure out where within those details we can compress.

The language of linear algebra allows us to express a solution by way of analogy: Vector spaces possessing more than one basis allow for a *change of basis*, this can be considered analogous to a reencoding of a text. The idea then is if we look at the "basis" of primitive functions that are used to construct a given composite, we can search for an alternative basis, or at least ask if one exists. When such alternatives exist, we can then research as to whether or not they compress the representation of the function we're interested in.

Beyond this starter strategy, how can we compress when it comes to non-primitive functions?

functional compression

Although I don't deny that there may be a diversity of emergent patterns that could be classified as satisfying an intuitive understanding of functional compression, there is one pattern in particular which is worth introducing here.

With this said it might be best to demonstrate this pattern in the context of real valued functions. Furthermore, in this discourse it might be clearer to refer to such functions as *grammatical forms*, while functions that act on such forms as *operators*. If this is the case, and we consider well known arithmetic and algebraic forms as our primitives, how would we then navigate the combinatorial inventory of all such grammar? A starting place is to return to the idea of a successor function.

We couldn't navigate the whole of this grammar space with successor operators, but they would at least let us navigate this space along single streams of direction:

$$g_0,\ g_1,\ g_2,\ \cdots$$

this is to say:

$$\begin{aligned} \operatorname{succ}^e(g_0) &= g_1 \\ \operatorname{succ}^e(g_1) &= g_2 \\ \operatorname{succ}^e(g_2) &= g_3 \\ &\vdots \end{aligned}$$

Operators such as these would then be called *enumerative successors*, being denoted as succ^e.

As for an actual example, we could use an enumerative successor to navigate the following points of grammar in this space:

$$\frac{1-x^0}{1-x},\quad \frac{1-x^1}{1-x},\quad \frac{1-x^2}{1-x},\quad \cdots$$

this being the well known *geometric series*.

I chose this series as it also has another important property when it comes to functional compression: For each term in the above sequence, there is a composition of algebraic operators that allow us to navigate to the next term. I have chosen to call such operators *iterative successors*, with denotation succ^i. This is to say:

$$\begin{aligned} \operatorname{succ}^i\left(\tfrac{1-x^n}{1-x}\right) &:= \tfrac{1-x^n}{1-x} + x^n \\ &= \tfrac{1-x^n}{1-x} + \tfrac{1-x}{1-x}\cdot x^n \\ &= \tfrac{1-x^n}{1-x} + \tfrac{x^n-x^{n+1}}{1-x} \\ &= \tfrac{1-x^{n+1}}{1-x} \\ &= \operatorname{succ}^e\left(\tfrac{1-x^n}{1-x}\right) \end{aligned}$$

at this point you may have qualms about a successor operator being defined in terms of the index of its current term:

$$\operatorname{succ}^i(g_n) := g_n + x^n$$

but this is nothing more than recursion, where computationally you would extract the n from g_n and use it to dispatch to the successor term accordingly. In this sense we've already seen such a successor by dispatch when we procedurally defined the successor functions for $\{0 \leq \mathcal{M}_3 \leq 4\}$.

At this point, if you're unclear as to the difference between enumerative successors and iterative successors: enumerative ones iterate the natural number index directly, while iterative ones iterate the whole grammatical form.

This example is our demonstration for the idea of *functional compression*: Iterating the natural number index is a lower cost than iterating the whole grammatical form. In terms of functional energy savings we've compressed the path required to navigate these grammar points.

Keep in mind that this form of compression can also lead to structural compression, for example consider the expression:

$$1 + x + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9$$

which isn't a term of the geometric series, but is "close" as it is only missing an x^2 term. This way we can apply basic algebra:

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 - x^2$$

and structurally compress this as the equivalent expression:

$$\frac{1 - x^{10}}{1 - x} - x^2$$

The subtlety here isn't our ability to structurally compress, for in that case we could replace the whole expression above with a single α symbol if we'd like. The point I'm trying to make from a methodological perspective is we don't value the geometric series as a grammatical pattern because it can structurally compress mathematical expressions, but because it can do so functionally.

computing science connection

In the world of computing science there is quite a lot of methodological overlap with math. Although it can be considered an *aside*, I thought it worth giving an example of this here.

In theory there is a pattern when it comes to defining functions called *tail recursion*. This is where you define a function recursively and only at the end of its definition does the recursive call show up. In lisp programming style we can define the factorial function as such:

```
(define (factorial prod num)              %    define num!
  (if (eq? num 0)                         %    if num == 0
   prod                                   %    return prod
   (factorial (* prod num) (- num 1))     %    else (recursively) call
  )                                       %     (factorial (prod * num) (num - 1))
)
```

In practice the value of putting function definitions into the tail recursion pattern is it allows for what's called *tail call optimization*. The idea is you can reuse the same memory variables/addresses/locations in calling the next iteration of this function because the output has the same type signature as the input (not to mention the previous input has served its purpose and can be forgotten without consequence).

The connection to functional compression comes when looking at our geometric series example: If you consider its grammatical form to have a type signature, you'll notice when we apply the iterative successor the type signature remains the same. It's tempting to intuit that there might be a connection here.

Unfortunately this turns out *not* to be the case: Best I can tell it's purely coincidental. Functional compression privileges self-similarity and slow (logarithmic) growth of the grammatical form as it iterates, and from this a fixed type signature ends up as consequence. Tail recursion on the other hand privileges composability, and it is for this reason it ends up having a fixed type signature.

So, yes, both end up demonstrating a fixed signature, but for different reasons. For example in the case of tail recursion self-similarity isn't even strictly required (coproduct types), and slow growth constraints within the grammatical form tend to be relaxed as the focus is on function application rather than function construction.

Although prioritized towards math, the methodologies discussed here can also be used to reason about computational design, and in fact do help clarify some patterns and paradigms seen in its literature. To reiterate this: When it comes to programming methodologies, functional compression is best used in defining the body of a function (regardles of module division), while tail recursion is better suited for defining function signatures for application.

navigational methodology

We've reached the end. This final section is more open ended, but a few words should be said about navigational methodology.

For starters, looking for generic patterns and methods to navigate arbitrary spaces is its own field of study, inevitably with its own set of methodologies. This is a pretty big claim, and the reason I am willing to make it is I have observed throughout mathematical literature that when mathematicians construct a new *type* (or set), there are effectively two base schemas to do so:

1. constructively: The instances [1] exist first, then we build the type to be defined as the full inventory of those objects.

2. conceptually: The type is defined first, then any potential instance can be validated as such by testing against the definition.

The consequence here is when a type (or set) is constructively defined, immediate navigation is less of an issue. In order to achieve construction one would implicitly need to navigate the objects to effectively list them out. In this case a basic system of navigation already exists. For example with the natural numbers $\mathbb{N}$ we could define them as a countable collection which is otherwise listed out using the successor function. The means to navigation is thus implied.

On the other hand if a type (or set) is conceptually defined, we can speak of *all* instances, but often there is no implied way to immediately navigate those instances. In this case navigation becomes its own field of study because all of a sudden we need to figure out how. In this case, the usual approach I've observed is through topology, where we research ideas of "nearness" and use them to ideally build metrizable spaces.

For example *continuous real-valued functions* are defined conceptually using Weierstraß's epsilons (ϵ) and deltas (δ), to which we then invest a lot of topological effort to figure out how best to navigate this space. Another example is the function space of μ-recursive (computable) functions, to which entire programming language grammars are devoted to navigating these grammatical forms.

With that said, I would suggest functional compression might offer an additional toolset when it comes to such topological considerations.

the shape of grammar

I'm willing to say whenever you have ideas of *nearness* within a space there's probably a topology hidden in the background. I will even conjecture that common mathematical grammar has underlying topologies we have yet to recognize. This is, if for no other reason, because we can reasonably speak of things such as: " $1+3x^2$ is symbolically closer to $3x^2$ than it is to $7-x+x^{12}$ " and mean it.

grammatical metrics

Another reason I feel there are underlying topologies for mathematical grammars comes from the current state of computer software: Programs such as **word2vec** in the tech industry use machine learning to map natural language words into vector spaces, thus providing distance functions (and geometries) that allow for surprisingly accurate language translation models.

I'm comfortable saying such an approach could be taken directly with mathematical grammar (mathematical words) as a proof of concept, but to be fair, the idea I'm trying to present here is that emergent properties of grammatical spaces such as functional compression could be used instead to outline definitions of topologies, these being a more natural fit for artificial languages such as math.

Finally, a potential application here depending on how we go about this could be to specify an analytic approach to designing code libraries, or function inventories. In the world of computing how do we standardize one implementation of a function over another? How do we compare constructive/navigational paths of function construction?

grammatical corpus

With the idea of functional compression we could also build a database of privileged grammar—a grammatical corpus—for reference.

Using the geometric series again here for clarity of example: Let's say you're working in a context where you recognize that the grammatical form x^n shows up frequently (polynomial contexts maybe?). Based on this corpus, we could look up this form (assuming we were unfamiliar with it) and find the geometric series to be a good compressor for the situation. This way, unless (or until) we have a better theory to organize these details, we can build up a combinatorial inventory classifying major grammar points within commonly modelled spaces.

semantics

Appealing now to linguistics, the idea of general *semantics* is very much about the relationships between grammar within a language: How the grammar relates, how constructs interact, that sort of thing. So if the grammar spaces of our formal languages have underlying topologies which relate the grammar to each other, I would conjecture that topology might offer a valuable model for semantics itself.

If nothing else an underlying topology for a given grammar space could at least lead the way to creating a comparison against another grammar space (and its underlying topology), and that alone I think is worth further research.

Thanks.

[1] Technically when constructively defining a type the objects used to build the type can't be called "instances" prior to the type definition.