# Stratica Theory:

# Or,
# How to design a code library,
# A mathematical approach

Daniel Nikpayuk

September 19, 2020



## Abstract

The intent of this essay is to informally describe a schema, called the *stratica schema*, that allows us to build and organize foundational or otherwise large code libraries in scalable ways.

## Philosophy

A brief introspection as to the nature of code libraries is a likely good place to start a philosophical discussion about library design. With that said, we might wish to assume some familiarity with code building and skip the basic question "what is a code library?" instead vying to ask: What do we seek when designing a code library?

It is part of the philosophy of this essay that the above question is in fact the wrong one to ask, or at least the wrong first one to ask. The more important question should be:

*What do we seek when designing a programming language?*

The short answer to this particular question is a non-affirming one: There are a lot things we might seek when designing a programming language, which is to say there are in fact a multitude of reasons to design a language—many of which are context specific. As such, a common set of design specifications might not be entirely possible.

Regarding such complexity, I am willing to claim that general programming languages should at least still privilege the following five values:

1. **Object Existence:** A language should be sufficiently potent to express all objects of interest.
2. **Name Uniqueness:** We should be able to prevent name collisions when naming the objects we build.
3. **Syntactic Validation:** We should be able to test the validity of the grammar we use to build objects.
4. **Semantic Verification:** When our grammar is valid, we should further be able to test that our program does what we intend.

5. **Heuristic Inference:** When the cost of proving code becomes untenable, we should still be able to test it under additional but otherwise weaker constraints.

What makes these these five values relevant is that we now have an answer to our above question in the positive. A question still remains though: How does such a value-system shape our philosophy when designing its libraries?

Let's start by pretending we were given a new programming language with bare minimum constructs. If we were to build a general library for it from first principles we would want to not only write code that supports the above values, but also common utilities that assist in user tasks. In either or both cases, after successfully writing such code we would then find ourselves with a large source body in need of organization.

What do we seek when designing a code library? We seek schemas for *organizational design* which additionally support the above programming language value-system in the following ways:

- We would want to be sure the organizational design does not **contradict** the above value-system.
- We would want to be sure the organizational design does not **weaken** the above value-system.[1]

As for organizational design itself? It, like our question about language design, also suffers from complexity: There is always more than one way, and anything that could be considered a "best way" would likely be context specific. In anycase, without further contextual information the only additional constraint for our organizational schemas at this point are for them to be **scalable**.

The philosophy of this essay then is that of an organizational design schema, but under the two constraints that it supports the above value-system, and is scalable.

# Methodology

In this section we will go over the prerequisite *methods* (and their logics) used within this essay to implement the above philosophy.

## Consistency

We first need to talk about ideas of *consistency* as they inform everything else which follows.

We start with theory. It is well known in the theory of logic that any formal mathematical/computational language (that is also sufficiently expressive) is unable to prove its own consistency. This is to say we can't use a language to self-prove that it has no contradictions. As a consequence, we must always defer the proof-of-consistency of a language to the *assumed* consistency of another. If we take this to its logical extreme, this unfortunately has the potential to be a "turtles all the way down" infinite regress sort of situation.

In practice what this means is we can't actually guarantee the consistency of the language we use—or any library built from it. This is disconcerting for sure, but it's best not to take a fatalistic attitude about it. Once we accept this realization, we can then choose to actively mitigate these complexities.

Within the realm of mathematics such complexity has been historically dealt with using what are known as *foundational languages*. For example, the first modern foundational language of mathematics is Set Theory which takes an *axiomatic* approach to mitigating consistency: We assume a handful of axioms which tell us about "sets" and "membership", and derive every other truth from these assumptions.[2] In fact Set Theory has been quite successful in supporting the development of many branches of math as the concept of a *set* makes for an intuitive language for *production semantics*.

Unfortunately, using sets to prove consistency at higher levels such as homotopy theory has become less tenable. More recent foundations such as Category Theory and Type Theory have responded to such limitations by baking a lot more of their consistency semantics into their initial definitions—much more so than Set Theory did. In turn, although research is ongoing, these theories are now providing formal languages which scale better within their consistency mitigations.

In any case, these are the pure mathematical theories, but even in practical technologies the problem of consistency shows up: For example, and notable here, each point within the previous list of programming language values is a known consistency problem: existence, uniqueness, validation, verification, inference.

[1] From a less formal (more heuristic) design perspective *contradiction* and *weakening* aren't always the same thing.

[2] Keep in mind, the methods of logical proof derivation are also assumptions within such a framework.

Given such designs, both theoretical and practical, the main question worth asking about the method of consistency becomes:

*If such consistency issues are dealt with at the language level, what is left to consider at the library level?*

The answer: Proofs and tests of consistency are aesthetic, artistic, and beautiful, but they are also *not free*—they are computational and they have costs. A proof might be known to be theoretically possible, but what if in practice it is so computationally expensive it is intractable or otherwise impractical? Schemas for scalable library organization might not play the prime role in mitigating theoretical consistency, but they still play important roles in mitigating consistency costs.

With this particular background out of the way, let's turn to the preferred consistency methods we'll use in the remainder.

### Dependency Mitigation

The first major method we will adhere to is what's known as **dependency minimalism**, which overlaps with Set Theory's *axiomatic* approach: We design our library to make as few consistency assumptions as possible, and derive the rest of the library's objects through its atomic ones. Taking this line of thinking further, we seek to find *minimal* or *shortest path* constructions from such atoms: The fewer the dependencies, the simpler the consistency proofs are for compound objects, the more likely the library will scale at cost.

### Abstraction Mitigation

The second major method we will adhere to is what's known as **expressivity of abstraction**, which is something we already take for granted in everyday programming languages—but still needs to be explicitly stated here. Basically, it just means our ability to abstract out and represent common patterns. Such expressivity is in support of dependency minimalism: When it's done well, it allows us to refactor and reuse pattern abstractions which in turn makes our library smaller.

It should be noted then that however we organize our library we will not only need room in it for specific objects, but for their abstractions as well.

## Modelling

Now that the general consistency methods have been introduced, let's discuss how we intend to specifically support our previously listed value-system.

In actuality, this essay only seeks to directly mitigate the first two value problems in the list: *object existence* and *name uniqueness*. As noted earlier, a programming language should itself be expected to take care of these issues theoretically, so we shouldn't have to consider their solutions in full. What's more, our schema will in fact retain greater flexibility for domain specific uses if we support but otherwise defer the other mitigations to the architect's locale.

As for the two value problems we will be mitigating, let's start with object existence. The method used to solve this problem is known as **modelling**, though as a concept I would prefer to introduce it by referring to [1], which itself quotes the following passage:

> The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.
>
> John Locke, An Essay Concerning Human Understanding (1690)

To reiterate what is being said—but in more progressive terms[3]—Locke states that we build ideas as follows:

[3]The intention here is to take a critical and *careful* reading of this passage, ideally separating out connotations concerning problematic legal philosophies of "property" at the time the passage was written. See Cory Doctorow's "Terra Nullius" for a good introduction: https://locusmag.com/2019/03/cory-doctorow-terra-nullius/

1. We define atomic ideas.
2. We define ways to combine these atoms into compounds.
3. We define ways to compare across all such ideas.
4. We define ways to abstract out common patterns across all such ideas.

This in its effect is the method of modelling we'll be using here. In particular though, in practice these listed steps are often interleaved and interwoven to create new models, we on the other hand will want to **orthogonalize** these steps as much as possible. This is to say we seek to orthogonalize our models toward being either *constructive* or *constrictive*.

### Constructive and Constrictive Modelling

Constructive models are ones which are **combinatorial**, they are the *atomic* and *compound* ideas discussed in the above passage by Locke. They are combinatorial in the sense that they are formed from *all possible combinations*. In this essay, when the context is clear we will also refer to them as *primary* models. In math, a common example of a primary model is that of a *span* defined within the context of a *vector space*, which is created from a basis by grouping all of its *linear combinations*:

$$a_1 \mathbf{v}_1 \ + \ \ldots \ + \ a_n \mathbf{v}_n$$

Constrictive models on the other hand are created out of primaries through **filtering**, or restricting an already constructed space. In this essay, when the context is clear we will refer to constrictives as *secondary* models.[4] In Set Theory such filtering is known as *subsetting*:

$$\{ \, x \in S \mid P(x) \, \}$$

For example, in terms of specific subsetting we could take the primary model $\mathbb{R}^2$ and subset it to become the classical secondary model known as the unit closed disk:

$$\mathcal{D} \ := \ \{ \ (x, y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1 \ \}$$

### Combinatorial Limitations

There's a subtle relationship between construction and combination we need to talk about if we're going to use them as methods.

While exploring mathematical models of infinite number systems, it has been discovered that universal combinations are a powerful tool to construct models, but they come at a cost: **navigation**. In practice we find many reasonable spaces where we start with a *navigable* source, we use it to combinatorially construct a derived model, and then find out the new space itself lacks a clear means of navigation. This is to say, combinatorial construction does not necessarily preserve the navigational properties of its source.

Some of the most notable models come in the form of *powersets*[5] of infinite sets. For example, if we take the powerset $\mathbb{P}(\mathbb{N})$ of the natural numbers $\mathbb{N}$, how do we access its elements? It's known the "size" or *cardinality* of this powerset is at least the continuum (the size of the real numbers $\mathbb{R}$), so it's tempting to think we could equip it with methods similar to those we use to navigate the reals. Unfortunately, as far as I know there's no intuitive way to navigate all the members of $\mathbb{P}(\mathbb{N})$, either through an ordering relation, or a topology: Does such a set have a positional system similar to the reals? Does it even have enumerators similar to the successor function $\text{succ}(n)$ for the naturals?

To put this in more philosophical terms, let's say we're given an arbitrary modelling space, our question becomes: What guarantees do we have that we know how to *access* all of its objects?

[4] Strictly speaking a primary model such as a span can always be created after the fact by filtering, one only need find a *superspace* containing it. The intention of the idea of a primary is that of *emergent construction*. In turn, a primary can always be defined as a secondary within a larger model, but not all secondaries can be defined as constructive! It should also be noted that within this philosophy of modelling, even with a plethora of secondary spaces there must always be at least one primary in order apply such filtering in the first place.

[5] Let $S$ be some set, the powerset $\mathbb{P}(S)$ is the set of all subsets of $S$. In this essay's modelling terminology, this is a primary model built from the combinatorial construction of certain secondary models (subsets).

### Navigational Ecologies

We're not yet ready to introduce the stratica schema, even still, I will say here that it incorporates combinatorial modelling to ensure object existence. As stated above this comes at a navigational cost—meaning we intend to solve one problem and in the process we'll introduce another. Given this foresight, what's to be done about it?

To restate this problem specifically: In the process of constructing modelling spaces we will end up creating subspaces which have no natural means of navigation. As for the solution, although I hadn't mentioned it in our combinatorial limitations discussion, the default technical solution here is known as *the well-ordering theorem*[6] which states we can always create orderings for the objects of such spaces. In particular, these orderings actually behave a lot like the ordering of the natural numbers $\mathbb{N}$: This is to say, we can in a sense *list out* any modelling space, even ones with cardinalities larger than $\mathbb{N}$.

The major issue with well-ordering as a solution is that it still doesn't tell us how to order a given model. We have to decide that much for ourselves, effectively choosing from all possible permutations of the objects we want listed out. To mitigate this particular navigational problem—as well as a few others we'll come across—we extend our method of *dependency minimalism* to that of **narrative minimalism**.

The idea is to organize navigational ambiguities and disorders through narrative storytelling. Although such is formally outside the scope of what is known as "mathematics", we will supplement mathematical rigour with our own human perspectives in order to navigate the complex maze of dependency paths within our libraries—done according to human relatable narrative designs.

This is a *heuristic* mind you, yet with that said there is much evidence throughout the stories of humanity to support the idea that this form of heuristics is one which we're good at. At the same time, this method doesn't scale indefinitely, and in the long run other approaches will still be needed. I mention this method here because it also shouldn't be overlooked or taken for granted at the human level. Our library should account for it as well, or at least not prevent it from being supported.

### Modelling Functions as Text

Up to this point we have yet to account for a special type of object which pervades any computational theory: That of the mathematical **function**. We use functions in math and coding all the time, but what is a function? really? and how do we model such an object?

It could be said that function design suffers from complexity[7] in that all known models are either properly contained within the space of *all* halting functions, or they properly contain this space—they contain all computable functions, but they also contain objects called *partial functions* which don't always halt. In turn, we've settled for $\mu$-recursive or Turing complete *primary models* as the default superspaces—for a lack of more accurate modelling techniques.

Without getting into these complexities, let's deconstruct one aspect of the nature of functions—something to take with us—but from a philosophical perspective: To get the ball rolling as they say, I would claim here it is a common intuition among function users that functions hold a connotation of **animacy**. This is to say that functions are able to take objects and "do" or "act" upon them. What do you think, do you agree?

In anycase, even though this sort intuition can lead to relevant narratives and discourses about the deeper nature of functions in general, for the style of library modelling within this essay I find it's actually better to interpret functions in the opposite way, as *inanimate*. As for animacy, we vie to transfer the weight of such semantic associations to what I would term **evaluators**.

The idea here is best expressed through analogy: In this essay we will interpret functions as *texts*—as in literature. If this sounds like a bit of a stretch, I would argue it is already common practice in both math and the world of programming to view functions this way. Source code for example is really just a collection of functions (or one big function), notably stored, or represented on the computer as a *text file*.

It could be suggested this is just a quaint anecdote, but I would further argue there are many benefits to such a change in perspective. For starters, it actually aligns quite nicely with existing theories: For example in Type Theory one has function types, but there's nothing immediate in its theory as to how one uses these types to model

[6] The well-ordering theorem is known to be equivalent to the *axiom of choice*, a potential axiom of Set Theory which historically has been controversial, though now is widely accepted. For our purposes this is a non-issue either way as our real world libraries will only ever be finite in size—we could in consequence restrict ourselves to the countable axiom of choice for which few if any mathematicians take issue.

[7] A recurring theme in this essay.

specific functions. There's the Lambda Calculus for one, and although it is of great theoretical importance it's also not exactly the most intuitive approach in navigating toward individual library functions with intended semantic behaviours. If on the other hand we accept a textual interpretation—saying that structural types have the potential to represent functions—such modelling provides a much clearer path as it is already familiar to what coders do now.

In these cases any instance of a structural type which can be equipped with an evaluator can then be considered a text, otherwise it is either a regular data structure or at best the representation of a partial function. Moreover, by classifying instances of types as texts by means of specific evaluators, we can further define the notion of a *genre* to which such texts belongs. This is to say: Genres are defined as types equipped with specific evaluators. In particular, a known evaluator would then create a *reading* of a given text.

The practical benefit in viewing functions this way is that they more naturally fit into the idea of a constructive modelling space. As well, by separating out evaluators from functions, we again adhere to the idea of dependency minimalism. All functions would be built from evaluators, and even some evaluators would themselves be built from simpler evaluators. As consequence, we would then reduce the total number of evaluators assumed to be consistent.

Finally, by isolating out evaluators from functions we can also more clearly build evaluator algebras, which might lend itself to a computational theory of *interpretation*, or *hermeneutics*. Such a theory could even be used more expressively to model complex evaluators intended to read complex texts.[8]

### Nomenclature

At last, we can now move past existence issue methodology, and address the *uniqueness* issue.

The solution of this problem happens to be much simpler, as are its methods: By taking a combinatorial approach to our modelling we have already laid the foundations. It becomes our intention then to extend existing methodologies so as to guarantee the prevention of colliding names as well.

#### Locations as Names

In particular, we intend to design our stratica schema with its own *landscape*, one which implies the idea of location based on "navigational paths" determined by *how and when* a given modelling object was constructed. As for the combinatorial limitation, any navigational ambiguity present can be resolved by assuming a well-ordering. We will observe that this leads to a unique universal name for each object in the modelling space.

Beyond this, we will also want "convenience" names: The systematic approach of using locations as names is effective, but often lacks intuition. It might be acceptable at the architectural level of a library, but we will want friendlier names for general (non-architect) users. With this in mind, let's reorient our perspective: If we were to now assume the possibility of name collisions, the problem then becomes one of resolution rather than prevention. To mitigate such scenarios, the general technique would be to use some form of what's known as a *tie-breaking* strategy.

Finally, within the scope of our organizational schemas the most natural tie-breaking strategy would likely be to introduce a **module** paradigm and tether our name resolutions to the modules in which they belong. This will be our approach, and we'll get to it shortly, otherwise this concludes the methodology section of this essay.

## The Stratica Schema

The stratica schema consists of three models.

The first model solves the existence and uniqueness issues, but with three caveats: 1) Object placement lacks narrative clarity, 2) The model does not allocate space for pattern abstractions, 3) The model does not encourage the idea or use of modules. These caveats are the source of the other two models.

The second model in this schema is provided in particular to address the second caveat: That of making room to place and locate abstractions. This model on its own it has a few subtleties we will need to discuss, but otherwise it parallels the first model quite faithfully. Beyond that, due to its *abstract* nature, the objects of this second model are what allow us to freely navigate the objects of the first—in effect this second model mediates between the first and the third.

The third model addresses the third caveat: That of modules, which are an important aspect of any library design! The origin of this model is to view the first as a reference—thus keeping our original solutions to the

[8] This could be a useful and intuitive way to approach compiler design.

existence and uniqueness issues—but to otherwise make a copy of this first model where we instead drop all of its structures, only keeping the "legacy" navigational content within the names. This is to say, we *flatten* the first model to derive the third. By starting over like this we can with the help of the second model introduce narrative storytelling into our design, thus encouraging modularity.[9]

## The Constructive Model

We will now present the first model in this schema, known as a **straticum**.[10]

Getting straight to it, a given straticum can be thought of as a collection of **universes**, where each universe is itself a collection of **languages**, each of which themselves are made up of modelling objects:

| | | | |
|---|---|---|---|
| **straticum** | ⋮ | ⋮ | ⋮ |
| | **universe - n** | ⋮ | ⋮ |
| | | **language - n/0** | objects - $n/0$ |
| | ⋮ | ⋮ | ⋮ |
| | **universe - 1** | ⋮ | ⋮ |
| | | **language - 1/0** | objects - $1/0$ |
| | **universe - 0** | ⋮ | ⋮ |
| | | **language - 0/k** | objects - $0/k$ |
| | | ⋮ | ⋮ |
| | | **language - 0/2** | objects - $0/2$ |
| | | **language - 0/1** | objects - $0/1$ |
| | | **language - 0/0** | objects - $0/0$ |

Upon first glance the nesting depth in this diagram can be considered fairly rich,[11] but otherwise the structural design is exactly as straightforward as it appears. Beyond that, as a straticum is meant to be constructed we will want to start by populating its languages. We can assume the initial language in this model from some independent model,[12] or we can start from scratch. In the latter case we would then introduce an initial object of our making:

$$\text{object - } 0/0/0$$

Notice its navigational path (0/0/0) tells us (reading from left-to-right) this object is in the *zeroth* universe, then in the *zeroth* language, and is designated specifically as the *zeroth* object. Think of it like reading a **url** address when accessing the internet, or your computer's filesystem. In anycase, this may be the proper name, but given that such a navigational *pathname* lacks intuitive appeal, we will also use the following short name for convenience:

$$\mathbf{ob}_{\varnothing}$$

It should be noted that as we have no way to construct this object from any other, we must *assume* it to exist. By this paradigm it is also the only object in its language. What's more, this exact object whatever it may be is in fact

[9] This literary inspired approach additionally becomes a stepping stone for mitigating the validity, verification, and inference consistency issues at the library level, though as is a common theme in this essay the finalization of such details remains for the architect to determine.

[10] The word straticum is invented, otherwise it is derived from the verb "stratify" and is meant to carry connotations of stratification. This is also where the name *stratica* schema comes from.

[11] In the above diagram the universes and languages are indexed with natural numbers implying they are countable. I want it to be clear that making such a restriction (or not) is an open problem. Either way, for the purposes of this essay we do not explore any possibility other than countable collections of universes and languages within these models.

[12] In this case, we would need to guarantee the exterior model's consistencies in advance.

local to its given straticum, which is to say if we start from scratch with another straticum there's no requirement for them to have the same initial object.

From here we build other objects to populate other languages, and we do so in a recursive manner. In order to do this, we must *assume* some construction operator:

$$\mathbf{op}^{\star}$$

This operator like the initial object is also local to its given straticum. Furthermore, and to give it a more intuitive appeal, we will in practice find such constructors are often "grouping" operators.

To construct the next language we now take the objects of the currently constructed one and combinatorially apply our assumed operator to them:

$$\textbf{language - 0/1} \quad \sim \quad \mathbf{op}^{\star}(\{\ \textit{all}\ \text{objects - 0/0}\ \})$$

In the current case as we've started from scratch the only object of our initial language is $\mathbf{ob}_{\varnothing}$, and so the next language for us becomes:

$$\textbf{language - 0/1} \quad \sim \quad \{\ \mathbf{op}^{\star}(\mathbf{ob}_{\varnothing})\ \}$$

Either way, I use the tilde ($\sim$) operator here to casually mean "defined as" rather than the standard ($:=$) symbol. After all, this description isn't actually meant to be formal, it's only to convey the concept.

To get the next language in the sequence, we continue applying our construction operator:

$$\textbf{language - 0/2} \quad \sim \quad \mathbf{op}^{\star}(\{\ \textit{all}\ \text{objects - 0/1}\ \})$$

The pattern then is to reapply this process ad infinitum to get the remaining languages—within this first universe.

From here, as we've now reached the limit of our first universe we build the objects of the *next* one by collecting together *all* of the current universe's objects as if they're now part of one collection. This is to say, we flatten the language structures and start over as if all the objects are now in one big language. This then becomes the first language of the second universe.

Following that, we again continue with the same combinatorial construction process we started out with, eventually building all the languages of the second universe. From there the cycle repeats to build the next universe, then the one after that, and so on and so on, until we have built all possible universes. Thus our straticum becomes complete! Unfortunately, we're not quite done—we now have to address how this model solves the *existence* and *uniqueness* problems.

In terms of existence, proofs are in actuality always context specific. With that said, the framework presented here equipped with the right initial object and the right grouping operator is enough to demonstrate models of set theory sufficient for most (if not all) of mathematics. Admittedly, such will not be proven in this essay as it takes us too far away from the intent, but I will give an outline in the application section.

As for uniqueness? To be clear, when we say "uniqueness of name" we don't mean that every object has a unique name, rather we mean every name has a unique object. This is what prevents *name collisions*, which then prevents ambiguity. As for such consistency, it is the consequence a straticum's structure: We already have unique *universe* and unique *language* pathnames, we only need choose a well-ordering or some other navigational system satisfying our ideal to then also have a unique *object* pathname. Putting these together: For each model pathname we are able to associate with it exactly one modelling object, and no modelling objects are left out.

This concludes the straticum model.

Before heading on there's one subtlety worth discussing about the idea of "uniqueness": Namely, there are no assumptions made in this model's specs that tell us whether or not the modelling objects themselves should be unique. What's more, if we look closer at the model's construction, there are in fact many opportunities to introduce object **duplication**:

1. There's no restriction placed on the construction operator saying it "must only create new objects". Hence, every time the operator is used it could potentially be introducing duplicates.

2. Every time we flatten a universe and use it as the starting point of the next universe we are also potentially introducing duplications.

Such subtleties are not in and of themselves issues for our purposes, but I will advise that care should be taken when implementing any related practical policy: It's very much a "pass by value" or "pass by reference" sort of situation, where either has its strengths and weaknesses across contexts.

## The Abstractive Model

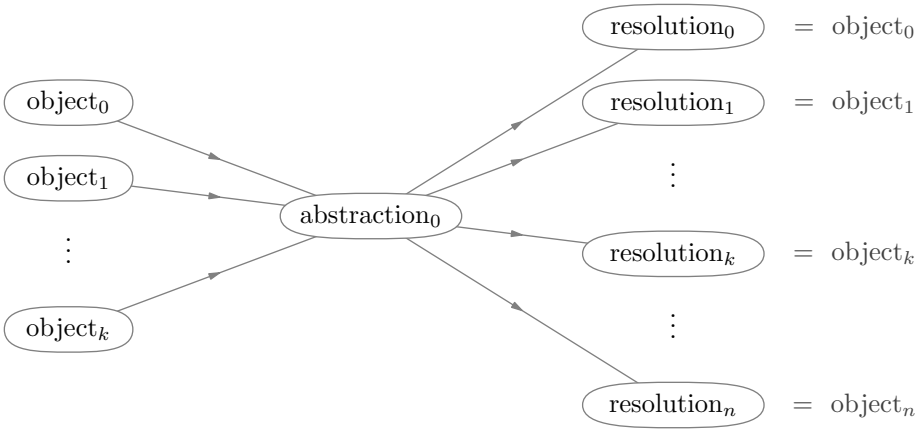
Every straticum has a companion model, called its **patronum**.[13]

The patronum is where we place the patterns that we are able to recognize and abstract out from our straticum of interest. For the most part this is straightforward, but it's also not without its philosophical subtleties. For our use, there are two immediate issues to be brought up: 1) How do we express a given pattern abstraction within its patronum? 2) Where *exactly* do we locate a given pattern within its patronum?

The first subtlety has been historically addressed in different ways across different mathematical theories. In Zermelo-Fraenkel Set Theory we describe patterns by means of *predicate* logic. In Martin-Löf Type Theory such abstractions would be the *dependent* types and functions. In Category Theory it would likely be *representable* functors. Is there anything like a universal approach? The short answer is no. Still, if we step back and observe across these existing theories, we might recognize a few *meta-abstractions* they share such as the use of *variables*.

In general the syntactic approach isn't actually the way to go in finding commonalities. The better approach would be to take a semantic view, in which case each "language of patterns" could be said to describe what we'll call **specifications**.[14] To follow this, how then do we express specifications for patrona? Do we use an existing grammar from one of the above theories? Do we make our own? The answer to this is that any actual answer is language and context specific, meaning it's best to leave it for the library architect to decide.[15]

With that said, there is one important restriction we should require for any given patronum's *language of patterns* to have: We need to be able to "reverse" the abstraction process.




This is to say, it should be reflected in the grammar itself that for whatever source objects we use to create a given abstraction we should also be able to turn things around and take refinements which then *resolve* back to each source. Furthermore, as an extension of this logic it should be said to be quite reasonable that we may resolve toward new objects as well. After all, in practice when we abstract out a new pattern we don't necessarily look at every single source object in the process.

The second subtlety about these abstractive models is in regards to where we should locate the *spec* expressions within their respective patrona once suitable grammars have been chosen for their patterns. The good news is our answer to this is already somewhat familiar as it builds on our existing methodology regarding navigational and combinatorial limitations.

For starters, our patronum is expected to maintain the same navigational structure as our straticum, so its navigational paths will be determined as before:

$$\textbf{universe-}p_0\ /\ \textbf{language-}p_1\ /\ \textbf{object-}p_2$$

[13]The word *patronum* is the Latin accusative singular case of the word "patronus" which translates as "protector", or "patron". This word was chosen because it is also the root for the word *pattern* which is appropriate to its intended meaning. As for the accusative singular case, it translates as a direct object rather than subject—this is also appropriate to its intended meaning as the patronum is a tool to help build the third modelling space within this schema.

[14]We have used the term *specification* informally throughout this essay already, but here we're interpreting it to have a stricter meaning.

[15]Mind you *context specific* as a trope comes up often as an answer in this essay, but for our schema to be effective its design is required to balance generality with utility. Some *undefined behaviour* is expected.

Beyond this, since no immediate answer stands out as to where to place pattern expressions, we will take an incremental approach to our solution, starting with a naive first attempt.

A given pattern abstraction has its origins in its straticum, and so our most natural first attempt would be to take its specification and simply seek, find, and inventory all **resolutions** of it within the straticum. By doing things this way we can then declare the spec expression's navigable paths to parallel these same straticum locations, but within the patronum instead. Unfortunately there's an immediate problem here in that there may be many objects matching our pattern, and they would likely be located in many different places. Such a *locus* description of locations would in practice often be too complicated to be considered meaningful.

Our next attempt extends this idea, where we now take the locus of our spec expression and choose from it a representative as our desired pathname. The problem is, how do we choose such a representative? Ideally we would like to choose the *first* location within the straticum where the pattern matches, but we run into the combinatorial limitation problem again: Loci don't have natural orderings, and so there's not default here. As before, we can always fall back on a *well-ordering* for such navigation, or choose a localized system when we know one exists.

The other issue here is that even when we do come up with a system to place our pattern, we may end up with pathname ambiguity. This goes back to our *uniqueness of name* issue, though in the case of the patronum it happens due to the nature of pattern abstraction itself: A single object within the straticum may be representative of more than one abstraction. As we add further patterns to our patronum, we will by chance alone likely run into a name collision at some point. As there is currently nothing preventing this possibility, we will like before be required to make use of some sort of *tie-breaking* strategy.[16]

That's about it in terms of the patronum model. Before ending though, given the similarity in structure (and even patterning) between patrona and stratica, I would like to point out one additional similarity as well as a subtle but important difference. The similarity is that both the patronum and straticum are made up of modelling objects/expressions which are entirely **symbolic**. This in itself is something worth noting. As for the difference: A straticum's symbols are *concrete*, or **static**. A patronum's symbols on the other hand are intended to be **variable**.

## The Navigational Model

The navigational model is called the **modulum**, and is made up of objects called **modules**.

This model comes into play here because although the straticum and patronum are where we build modelling objects, as spaces they generally lack navigable intuition. For example in practice—be it mathematics or coding libraries—we as people tend to keep our theories organized by grouping together objects which we consider semantically related. As a specific example, take the common *pair type*: In a coding library we would usually place the structural definition of this object type together with its constructor, projection operators, and anything else we might want to keep close.

This much of course is known as **modular design**, and I would expect the reader to already be familiar with the paradigm. To put this in contrast with the navigable designs of the straticum and patronum, they as spaces are designed to instead place objects together based on a *constructive narrative*. Any objects in these spaces that we would consider related often aren't grouped anywhere near each other, and so to use these models as direct references for our objects would become mentally taxing as we'd be expected to remember every location—each and every time—when accessing specific objects of interest. This does not support our consistency mitigation efforts!

The solution then is to group together objects from both the patronum and straticum to become the modules that make up our modulum. In set theoretic terms we would flatten both the straticum and patronum structures and unite them into one big set,[17] for which we would then take its powerset, and that would be our modulum.

We end up going from a constructive narrative originating with the straticum to a *navigable* one with the modulum as we now seek to order its modules (subsets). In particular, since we are still privileging dependency minimalism, the most natural ordering would be to start with some chosen *origin* modules and use them to navigate toward (and order) other modules which depend on them. These in turn would be used as *source* modules of their own to then navigate to (and order) other modules still. And so on, and so on.

Given this line of thought, an alternative interpretation of our model is as a navigational *overlay* of the straticum. The mechanisms which make this possible are the pattern abstractions of the patronum: If we were to start at some

[16] One strategy that comes to mind is to give two or more expressions with a name collision the same location but then extend their names by one extra (but distinct) symbol. Such symbols would be handed out first come first serve.

[17] I note here that such a collection might be too large to be a set, in which case we default it to being what's called a *class*, which can be thought of as a set from a higher order modelling space.

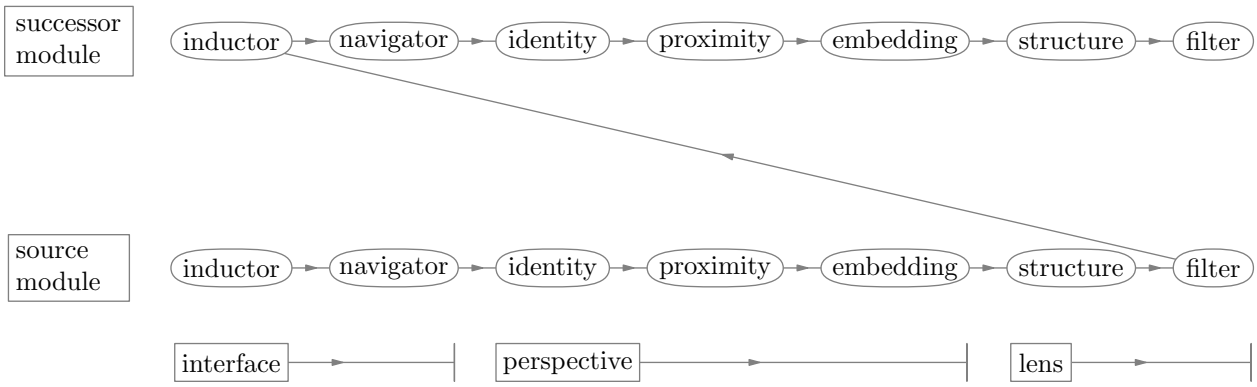
existing location in the straticum, and we sought to move to another location but using a path outside the existing navigational system, we could make such a path by first *abstracting* away from the current object, followed by moving across the abstraction space, finally we would *resolve* back down into another object of the original system.

Although this is the general idea of the modulum, things are admittedly still pretty vague, so for the remainder of this section we will be focusing on a specific paradigm for modulum building which not only ties together these module and overlay interpretations, but will also be used in the applications to follow.

### The Grouping Paradigm

Keeping with one of the major themes of this essay I will not claim that there's any best paradigm for modulum building in general. I will claim however that the one to be introduced here is worthwhile and widely applicable as it has been abstracted from patterns observed throughout various mathematical contexts. The main idea of it is to create modules by taking inventories of the tools needed to specifically "talk about" groupings of objects.

If we were to take a first principles narrative approach, this paradigm could be described along the following lines:




As there's a lot of information here, let's go over it.

We start with a *source module*, where we can assume it is already a grouping of some of the existing modelling objects. The intention is to use this space to build a narrative of *function inventories* which would allow us to navigate into more complex groupings of its objects, and inevitably other modules.

The first step in such a process is to inventory texts or functions which help us to classify this space's objects with logical precision. Such an inventory is called the **inductor** inventory, and its texts are the functions typically known as constructors or selectors. From there, just as we take for granted we can navigate the straticum and patronum as spaces, we also want to be able to take for granted the navigability of these module spaces. The inventory of texts that allows us to do this is called the **navigator** inventory, and its functions often come in the form of mathematical monoids, as well as other algebraic or arithmetic operators.

More broadly, the inductor and navigator inventories make up what I would call the *interface* of the module: They tell us what kind of objects are in the module and in particular how to access them. Following this, we would then turn to what I will call the *perspective* of the module: This is in the sense that its inventories provide the means to compare the objects of this space.

The first inventory here is called the **identity** inventory, and it's where we find functions that allow us to test ideas of identity—which is to say equality, inequality, and equivalence operators. Of course when two objects aren't equal (or equivalent), we may still wish to compare them in more refined ways: We might want to test ideas of *nearness* such as greater than, less than, or other ordering relations. The inventory holding these operators is called the **proximity**. As for the third inventory here, it is called the **embedding** and it is where we place the functions that allow us to make indirect comparisons of objects. For example we may have a homomorphism that allows us to compare within our local space, noting that such comparison is only achieved through the properties of the external structure to which the homomorphism maps.

The next family of inventories in this module make up the *lens*, whose functions allow us to navigate outward toward the other modules within this larger modelling space. The first inventory is called the **structure**, which is where we place *existential* grouping functions. They are existential in that they are not *universal*. Consider them as paralleling the constructive operators used to build the straticum, but in contrast they aren't required

to be used combinatorially. Following this is the **filter** inventory whose texts consist of logical functions such as predicate expressions. In a sense, the *structure*'s operators are oriented toward constructive modelling, while the *filter*'s operators are geared toward the constrictive variety.

From here, by assuming this narrative design in our source module, we should be able to use the operators within the structural and filtering inventories to move us to what we would consider a *successor module*, which has among its dependencies our current source. From there, the narrative process starts over, and again, and again, each time seeking out similar patterns of inventories for that module of focus. This then, is the overall idea of the paradigm.

There are of course some subtleties to take care of before heading on.

First, observing the above design—and given that we really only need the existential grouping operators to move to the next module—it should be asked: Is this rich "narrative" design really relevant? I argue it is in that it expresses a common thread of assumptions across grouping strategies. For example, in order to compare objects in general we tend to take it for granted that we must first be able to navigate and name those objects to begin with. As well, in order to eventually group objects we would need to be able to compare them—in atomic as well as compound tests—in order to semantically classify them as being together.

The second subtlety is that we need to address the possibility of overlapping inventories. For example, what if we find functions which we can suitably be classified as a navigator *and* an *embedding*? In such a case, we do what we've done in the past: We either allow for duplicates, or we come up with some tie-breaking strategy such as placing the given text in the first inventory to which it occurs within the above narrative.

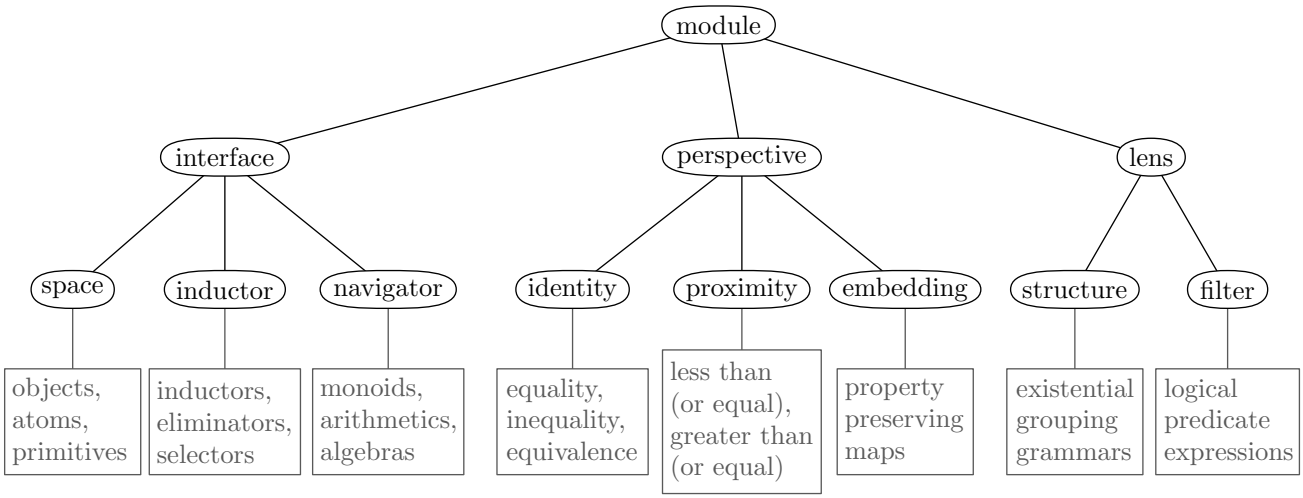
The third subtlety here comes in terms of needing ways to classify the objects of modules to help build intuitions for their organizational designs. I will say that in building specific narratives within my own applications I have found it worthwhile to be able to classify modular objects as adjectively aligning with the straticum or patronum. To this end, I introduce the following rules:

- If every component of a modular object is static, then the whole object can be considered **resolved**.
- If one or more of an object's components are variable, then the whole object should be considered **abstract**.

This classification scheme extends beyond single objects as well: If a module consists of resolved objects only it should also be considered resolved. If on the other hand it has at least one abstract object it should then be considered abstract.

The fourth and final subtlety of this model comes in how its narratives compare with that of its straticum. In particular, it is important to take note that this grouping paradigm is fundamentally *navigational* rather than *constructive*. To emphasize the difference, I will introduce the terms **narrative construction** and **narrative reenactment** to clarify when we are properly constructing new objects, and when (in contrast) we are merely navigating to ones which already exist. Such concepts are introduced to help us avoid dependency illusions such as the *circular definition problem*.

This concludes our section on the modulum. With that said, given the complexity of the above grouping paradigm let me represent it one more time, but here in what I hope is for the reader a more user-friendly referential format:



To that end, and in comparison to the previous figure, I have made explicit room for the objects of the module, which I now label as the **space**. I have further added intuitive descriptions to the object space and function inventories. Finally I have also tried to clarify the module's internal organization with the given tree structure.[18]

# Stratica In Action

In this short section I would like to briefly introduce two theoretically oriented stratica offering models for mathematics and computing science.

## A Mathematical Schema

We now turn our focus toward a stratica schema for modelling mathematics. In particular we look to model the objects of Set Theory, as it has been the main informant for the designs of this essay.

Let's now go over the schema definitions.

In terms of the straticum we can start with some preexisting but arbitrary space $\mathcal{S}$ as our initial language. We then choose the *powerset* operator $\mathbb{P}(.)$ as our straticum constructor. For convenience, let's assume the *union* operator exists as well, in which case we can reimplement the straticum definition in strictly set theoretic terms:

$$\mathbb{U}_\infty(\mathcal{S}) \quad := \quad \bigcup_{n\geq 0} \mathbb{U}_n(\mathcal{S})$$

This definition is incomplete, as we still need to define the universes. We do so as follows:

$$\begin{aligned} \mathbb{U}_n(\mathcal{S}) &:= \bigcup_{k\geq 0} \mathbb{P}^k(\mathbb{U}_{n-1}(\mathcal{S})), \quad n \geq 1 \\ \mathbb{U}_0(\mathcal{S}) &:= \mathcal{S} \end{aligned}$$

where $\mathbb{P}^k$ is the $k$th powerset. Given this definition, each $\mathbb{P}^k$ is then the $k$th language within its universe. Beyond this, and compared with the original definition of a straticum, note that $\mathbb{U}_0(\mathcal{S})$ is actually redundant. It is added to simplify the above *recursive* definition.

In anycase, I won't prove the properties of this space here, but I have already shown in [2] that the operators of set *union*, *intersection*, *difference* are closed within this straticum. Also, although I didn't think to prove it at the time, it requires only a bit of extra effort to recognize the powerset operator is also closed. This much at least can be more clearly witnessed with the following identity:[19]

$$\mathbb{U}_\infty(\mathcal{S}) \quad = \quad \bigcup_{k,n\geq 0} \mathbb{P}^k\mathbb{U}_n(\mathcal{S})$$

Admittedly these don't satisfy all the axioms of set theory, but they already satisfy quite a few of them. As for specific mathematical objects of interest, we already have access to the major number systems: The key is in how set theory defines pairs. For example if we were to assume the natural numbers in the meantime, we could then define its pairs as follows:

$$\begin{aligned} a, b &\in \mathbb{N} \\ (a, b) &:= \{a, \{a, b\}\} \end{aligned}$$

In this case we know such defined objects exist because we can locate them as belonging to $\mathbb{P}^1\mathbb{U}_1(\mathbb{N})$.

[18] As an auxiliary point, I would add that the *interface, perspective, lens* families aren't strictly necessary when implementing this design, they're there more for conceptual clarity. With that being said, I find within my own libraries I do prefer to retain these tags if only to keep things organized in my own mind. If you do decide to use these classifiers, I recommend tagging them as *partitions*, to which the umbrella term for the space + inventories would then be *divisions*.

[19] In fact the original name *straticum* derives from this structure due to its stratified representation.

If we then extend such pairings, it readily follows we can locate other relevant objects needed to construct the classical number systems of math:

$$\begin{aligned}
[(a,b)] &\in \mathbb{P}^2\mathbb{U}_1(\mathbb{N}) && \text{// equivalence class} \\
\{[(a,b)]\} &\in \mathbb{P}^3\mathbb{U}_1(\mathbb{N}) && \text{// equivalence relation} \\
\mathbb{Z} &\in \mathbb{P}^3\mathbb{U}_1(\mathbb{N}) \\
\mathbb{Q} &\in \mathbb{P}^3\mathbb{U}_2(\mathbb{N}) \\
\mathbb{R} &\in \mathbb{P}^3\mathbb{U}_3(\mathbb{N})
\end{aligned}$$

The notation used within this figure is actually misleading. The straticum certainly contains modelling objects which we would consider representative of our classical number systems such as the naturals, reals, complex numbers, etc., but these objects should be thought of more as *implementations* of those systems.

As for the patronum of this schema, we would effectively populate it with expressions of predicate logic, which has been the backbone of classical set theoretic abstraction from the beginning. In fact the number systems discussed above are actually *specifications*, and as such they properly belong to the patronum. And since we're here, I will add that the abstract branches of math such as *groups, rings, fields, topological spaces* also belong to the patronum! I didn't get into it at the time, but patrona are made up not only of abstractions, but abstractions of abstractions, and so on—or rather, patrona are made up of specifications to varying degrees of refinement.

Finally, regarding the modulum, it just becomes a narrative storytelling of how we build mathematics through its theorems, proofs, lemmas, corollaries, etc.

As an addendum, it should be said that this mathematical schema is actually *incomplete* as we have not specified its initial language $\mathcal{S}$. To be fair, we could have started from scratch with:

$$\mathcal{S} \quad := \quad \{\, \varnothing \,\}$$

Things would have worked out fine, just everything gets shifted up a bit in terms of path locations. Actually, it's potentially even better than this: The straticum of this exact schema now coincides with the *Von Neumann universe* which is a classical way to model Set Theory. What's more, in our case if we decide we want to use this schema in the same way our only concern becomes whether or not our straticum is "large" enough for this purpose. The good news here is that even if it's not we can always extend it through repeated application of this very same schema until it is.

Regardless of such choices, the reason I did not take this approach is that I find it preferable to first create an independent *computational* schema, and then use a flattened version of that as our initial language $\mathcal{S}$. It is with this mind we continue on to the next schema.

## A Computational Schema

I won't go heavily into this schema, but I'll say that we build it by starting from scratch where we assume some initial computational object $\mathbf{ob}_\varnothing$. From its initial language we build all others using a constructive operator I would call **powerbind** which lets us effectively "bind" all combinations of the language's objects together:

$$\mathbb{B}(L) \quad := \quad \{\; w_1 : w_2 \;\mid\; \text{for all } w_1, w_2 \in L \;\}$$

It's tempting to rephrase or even reinterpret this operator as a "pairing" and in all honestly that is the basic idea behind it, but because the concept and terminology of a *pair* is so central to both math and computing I would rather reserve that particular word for those purposes and contexts.[20]

In terms of our computational schema, and given we only have a single object in our first language, our next language would be as follows:

$$\{\; \mathbf{ob}_\varnothing : \mathbf{ob}_\varnothing \;\}$$

[20]To be fair, the word "bind" is not entirely uncommon in existing computing literatures, but I'm taking the chance that its meaning here is different enough that no confusion will arise. For the record, I am also open to changing the word in the long run if a more suitable one comes along.

The language after that then becomes:

$$\{\ (\mathbf{ob}_{\varnothing} : \mathbf{ob}_{\varnothing}) : (\mathbf{ob}_{\varnothing} : \mathbf{ob}_{\varnothing})\ \}$$

while the language after that follows as:

$$\{\ [(\mathbf{ob}_{\varnothing} : \mathbf{ob}_{\varnothing}) : (\mathbf{ob}_{\varnothing} : \mathbf{ob}_{\varnothing})] : [(\mathbf{ob}_{\varnothing} : \mathbf{ob}_{\varnothing}) : (\mathbf{ob}_{\varnothing} : \mathbf{ob}_{\varnothing})]\ \}$$

Notice the pattern? Each of these languages has exactly one object!

All the languages of the zeroth universe are this way. In this alone we could already start to model the natural numbers $\mathbb{N}$, though as a computational language we would also want—and the earlier the better—a finite number of objects other than counting numbers. Unicode characters for example. The most natural implementation in ensuring such a symbol-set would be to interpret the objects at the front of this countable "list" as the respective non-numbers.

Following the first universe, the second is where we would see more complexity within object expressions. At the same time, some basic set theory tells us that the size or cardinality of this universe is countable like the first. Not too much else can be said about this model upfront, but I will say it only takes a bit of understanding of cardinal logic to determine that this pattern of *countability* propagates throughout these universes. Furthermore, due to another set theoretic result about the countable union of countable sets, it turns out the whole straticum is countably infinite as well.

| | | |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| **language - m/n** | $\mathbb{B}^n\mathbb{U}_m$ | expressions - m/n |
| ⋮ | ⋮ | ⋮ |
| **language - 1/0** | $\mathbb{B}^0\mathbb{U}_1$ | expressions - 1/0 |
| ⋮ | ⋮ | ⋮ |
| **language - 0/k** | $\mathbb{B}^k\mathbb{U}_0$ | expressions - 0/k |
| ⋮ | ⋮ | ⋮ |
| **language - 0/2** | $\mathbb{B}^2\mathbb{U}_0$ | expressions - 0/2 |
| **language - 0/1** | $\mathbb{B}^1\mathbb{U}_0$ | expressions - 0/1 |
| **language - 0/0** | $\mathbb{B}^0\mathbb{U}_0$ | expressions - 0/0 |

As for the patronum and pattern abstractions, if we look at any given universe and the objects within we tend to observe structurally symmetric "binary trees" as the general pattern. To then recognize and abstract out more dynamic structures (and texts) we would have to range over the entire straticum itself. In fact to even be able to define a *list* we'd have to enumerate across universes.

As for the modulum, this is where we get to reorganize the narrative for the better: Although initially a little awkward, it can be readily seen that all the computational objects we might be interested in should be in this schema. In terms of the modulum if we were to equip this space with a few base evaluators, the narrative design then follows quite naturally along a *type theoretic* trajectory.

## Application: Code Libraries

With our stratica schema sufficiently explained, and having two important examples under our belt, it is now time to put the pieces together and turn our attention to the organizational design of code libraries in general.

To that end, and to complete our basic designs, we need to determine the following:

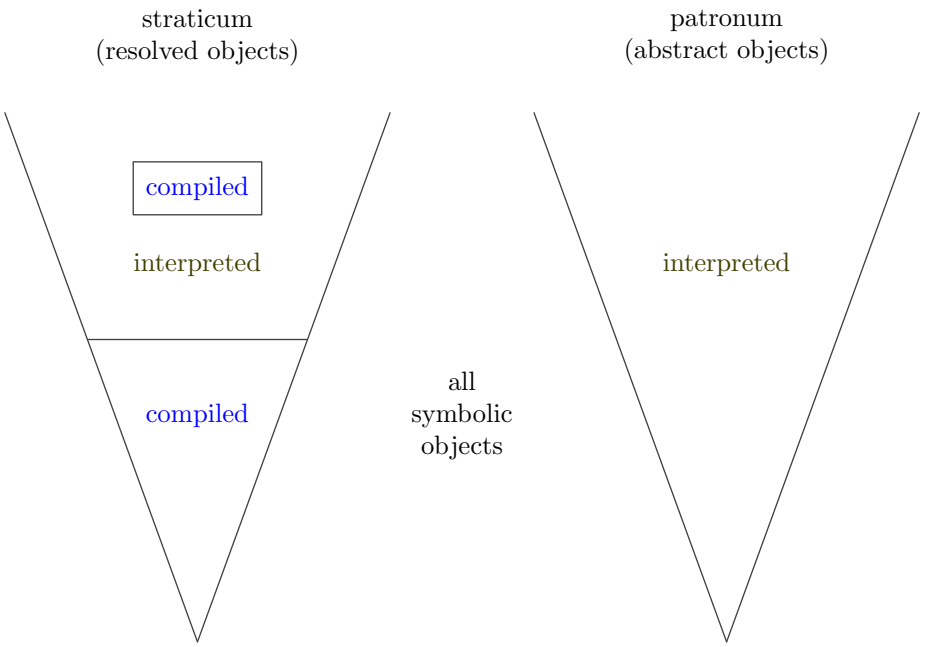
1. We assume that the stratica schema is independent of any given programming language. If this is the case, how do we interpret it within the context of a specific language?

2. I've made it explicit on several occassions that the modelling objects within our schema are entirely symbolic. If that's the case, how do we compile them into actual machine code?

The answer to the first of these questions is more subtle than the second, so let's start with the second.

## Machine Code

We conceptualize the objects of a given schema using the figure below:




Within the stratica schema itself, there are two competing narratives regarding object access: constructive, and navigational. In the context of object compilation, the constructive narrative is more appropriate in conceptualizing the relationships needed for object translation. As such, we only need look at the first two models of this design: The straticum and patronum.

Now, since we seek modelling objects which can be translated into machine code, we're effectively saying we seek objects which are resolved to being *computational*. Within the straticum we will call these objects **compiled**.[21] In particular, to ensure such objects exist we will initiate our design to be a computational schema similar or equivalent to the one in the previous section.

In the above figure then, we start with the left structure as representing the straticum—which fans upward and outward—and take its initial region to be made up of compiled objects. If you'll note, there's another region (boxed) above which also represents compiled objects. We'll discuss that shortly. As for achieving compiled object translation, we need to do so while still adhering to the methodological designs of this essay, such as dependency minimalism. Given this, we assume our language has a minimal *translation evaluator* which we can use to translate simple objects directly while also allowing us to build more complex translators using it as a component.

From here, while still within the straticum we would then begin to build the remaining objects which are notably *not* computational. To do this, and for our purposes, we can take the computational straticum as the base language of what now becomes a mathematical straticum.

As for an example of a resolved object which is not computational, take the exponential function:

$$e^x : \mathbb{R} \to \mathbb{R}_{>0} \quad , \quad \frac{d}{dx} e^x = e^x$$

This notation of course is misleading. This in fact is a specification, and little more. As we're dealing with the straticum we would instead like to have an implementation of this mathematical object. One well known *function*

[21] I use the word *compiled* in the sense that these objects *are to be* compiled in this context. The potential form of this word, *compilable*, might be more appropriate so as to not typecast these objects in terms of more general settings.

*as text* is as follows:[22]

$$\sum_{n\geq 0} \frac{x^n}{n!}$$

We take it for granted, but even implementation objects such as this one can only ever be symbolic! There might be individual values that when applied make such functions (pointwise) computational, and we certainly have theories of approximation which allows us to appoximate such functions using only computational functions, but otherwise these functions and their implementations as a whole are not computational.

Symbolic objects which are not computational are said to be **interpreted**. Thus, the remaining objects of the straticum are interpreted. Well, for the most part: The reason there is a secondary closed off box of compiled objects in the above figure's straticum is because some intepreted objects such as the above exponential function retain computational components, and as of yet I see no reason to deny such component objects the ability to become compiled. Another famous example by the way is the *Gamma function* which in general is not computational until we restrict it to a natural number subdomain, in which case it reduces to the *factorial function*.

We've classified the objects of the straticum, but how about the patronum? In the previous figure it is represented as the structure on the right, and given that it is entirely abstract its objects are classified as entirely interpreted. There's little else to be said about the objects of the patronum, except maybe to inquire a bit further about the nature of intepreted objects more generally: What is their relationship to compiled objects? What is their value within a programming language?

The name "intepreted" comes from the idea of an interpreter, which instead of taking source code and translating it to machine code for the operating system to run later (at the user's discretion), it translates directly and then asks the operating system directly to run it there and then—all the while still within its own running scope. Hence, if one does not wish to compile objects they can always interpret them. In effect these interpreted objects can be considered *strings* of symbols. This then requires interpreted objects to at least have computational algorithms for their symbolic manipulations. As for the relationship between compiled and interpreted objects? Compiled objects are a subclass of interpreted.

As for the value of interpreted objects within a general programming language? They exist to be used within the **proof assistant** component of a compiler or interpreter, though it's worth acknowledging that many programming languages deny or limit users direct access to such objects. Either way, they are used specifically to assist the compiler or interpreter in validating and even verifyng the consistencies of user designed objects, so that they know their translated source code will run the way they want.

As an aside, although there's nothing within this schema to set denial or limitation policies regarding interpreted objects, I will say that having greater access to these components at the architectural level is a better fit for any programming language that admits the mathematical schema introduced above. Such languages could then be used more generally for mathematical research and theorem verification. There are such mathematical proof assistants currently, but at the time of this essay being written I don't know of many assistants which are also compilers.

In anycase, all of this answers the second question of this section regarding machine translations. What about the first question? What about interpreting these schemas within the confines of actual programming languages?

## Language Fit

We've already answered this question in part: We construct the computational schema, and build the mathematical schema on top of it. When applying this schema to actual programming languages it's then a matter of fitting the schema's designs onto the grammar of the language.

In general, we can't assume a perfect translation or even a perfect fit, it's a matter of what grammatical constructs the given language offers us in the first place. I will say this though: The idea is to take grammar from the language and use it to represent the designs of our stratica schema. In doing so, when restricting ourselves to this grammar subset, we then take on the ideal properties of the schema's design. In turn, this sublanguage within the larger programming language ends up privileging our builtin well-behaved consistency designs.

More specifically, in terms of translation templates which *could* work for a wide variety of languages, it is likely more effective to focus on the navigational narrative of the modulum rather than the constructive narratives of the

[22]Strictly speaking, the particular grammatical form presented here isn't actually a resolved implementation. I use it in the place of one as a convenience as it is effectively *one step away* from becoming resolved. The issue with using a specific resolved object is that it would depend on the precise definition of its straticum, and such a rigorous treatment would take us too far away from the meaning of this example.

straticum and patronum. This being because the modulum offers us more flexibility in terms of how we can package together a library's objects. Taking this approach then, we would actually turn things around entirely from the constructive narrative: Instead of starting with compiled objects we now start with interpreted ones.

This inversion is quite natural given that interpreted—and more specifically abstract—objects are usually better suited for dependency minimalism. Given this, it becomes worthwhile to extend the "resolved" and "abstract" terminology currently applied to modular objects and modules:

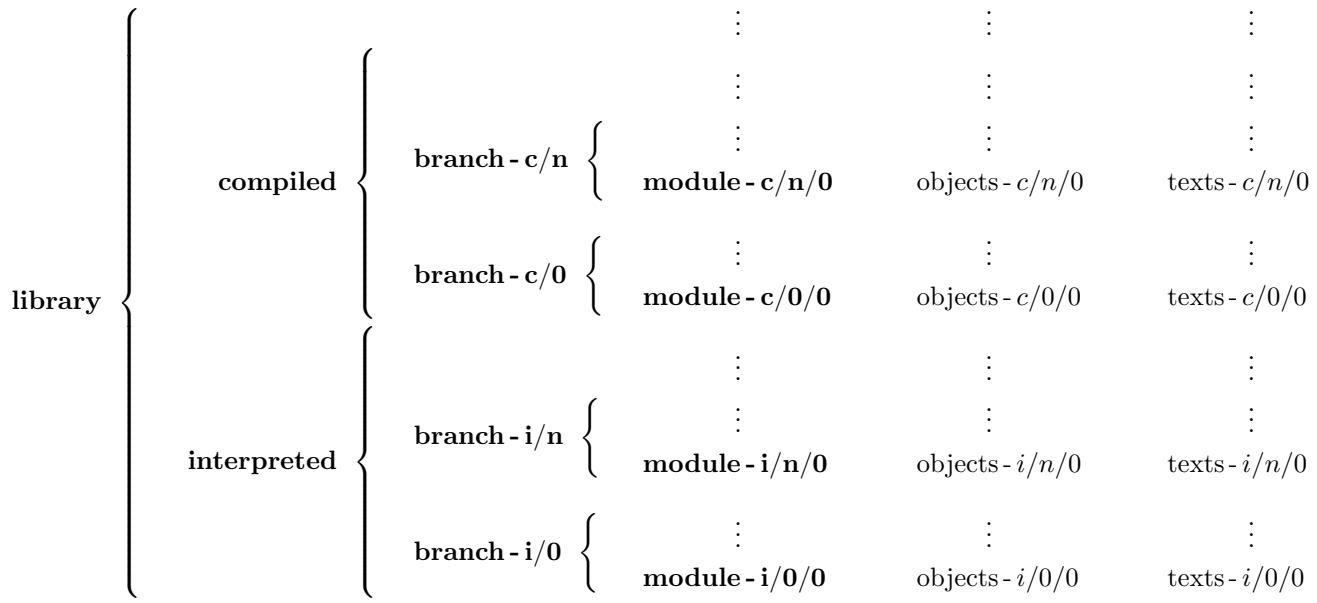
1. When an abstract object is close to becoming resolved, we will say it is **almost resolved**.

2. When an interpreted object (be it resolved or abstract) is close to becoming compiled we will say it is **almost compiled**.

Maybe this terminology is a little too transparent, but the point is to have at least something to differentiate objects within these subclasses.[23]

With all of this being said, we're still left with a lot of room for interpretation when it comes to the organization of actual library code. To assist in this endeavour I will conclude this section by offering some generic paradigms to fall back on when no others are preferred.

We start with a top down organizational design: We build upon our current framework of interpreted and compiled modules, but we extend it with an additional layer of narrative padding (so to speak) known as **branching**. This idea is reasonable as we expect quite a few individual modules to populate any given library that follows this schema. The terminology is inspired by and borrowed from the mathematical landscape, in particular observing the organizational design within mathematical literature. In anycase, our **library** is organized as follows:




The interpreted branches here for the most part would be branches of mathematics such as:

*type theory, groups, rings, fields, vector spaces, topological spaces, combinatorics, etc.*

The compiled branches on the other hand would more reasonably correspond to branches of machine *hardware* and related low-level software such as:

*screens, keyboards, touchpads, soundcards, memory, diskdrives, internet protocols, etc.*

[23] In terms of my own transparency, this policy come from personal experience as I have caught myself running around in narrative circles trying to clarify object and module dependencies before having these names.

The modules of these machine oriented branches would exist to facilitate cleaner translations to machine code, as well as improved interactions between computer hardware with respect to user coded programs. What's more, given the proof assistant underpinnings of this design, it wouldn't take much to also encode hardware specifications to use within these modules' consistency checks.

As for the narrative design of these modules, it makes sense to continue using the *grouping paradigm* of the previous section unless one has specific reason against it. In anycase, among the above noted branches there is one which stands out as having fundamental importance since all others will likely make use of its modules. This branch is that of *type theory*, and due to its importance I will here suggest the following narrative for its modules:[24]

$$\begin{array}{lllllllll} \text{judgment} & \longrightarrow & \text{(curried) function} & \longrightarrow & \text{equality} & \longrightarrow & \text{pair} & \longrightarrow & \text{copair} \\ & \longrightarrow & \text{boolean} & \longrightarrow & \text{stem} & \longrightarrow & \text{list} & \longrightarrow & \text{colist} \end{array}$$

I won't go heavily into their details, but the motivation is that ultimately everything hinges on the list type as it becomes the main component of many of the more interesting programming constructs used in practice. As such we would definitely want a *list* module in this branch. As for the other modules, taking inspiration from functional programming, I would argue that the following grammatical keywords are observed as common patterns across a large core of actual list functions:[25]

$$\text{eq?} \quad \text{cons} \quad \text{car} \quad \text{cdr} \quad \text{if}$$

This much alone already suggests we would need *equality*, *pair*, and *boolean* modules. We then prepend to this narrative basic *judgment* and *function* modules to achieve a base level of type theory itself. After this we would want to extend to a type theory with *algebraic data types*, to which we would then insert the *copair* and *colist* modules.

All that's left is the *stem* module. Its contents are the final prerequisite leading up to the *list* module. I will not get into the details here as they are given in [3], but the *stem* module contains what I call a *stem* operator which is key to defining a modular form of tail recursion required to otherwise build the major component of the *list* module: A 1-cycle list operator.

As for the *list* module and this 1-cycle operator, their details are provideed in [4]. In particular, the 1-cycle operator is designed to be a generic algorithm for acting on lists; it gets its name from having the constraint that it parses over a given list (at most) once. The value of such an operator in terms of a more general *list* module is that it can specialize into better known general list operators such as *map*, *fold*, *find*, and even *zip*. Finally, as an extension to this, the 1-cycle operator is also designed to be suitably composable such that it can then be used to build other more complex or higher cycle list operators.

That's about it when it comes to this schema and general programming languages. I would now like to mention a few design extensions before we finish, which is to say paradigms that don't contradict this schema, but are not directly supported by it either.

For starters, one reasonable way to extend modulum objects is to allow for alternative *implementations* of the same content. This is to say, we could allow for two modules to have the same name, but otherwise would be tagged as being different implementations. For example one might come up with a given module for the *copair* type[26] based on a set theoretic *disjoint union* implementation. Then again, one might also implement it based on a category theoretic definition in which no internal structure is assumed. Granted it's possible to create a single module as having both implementations, but the point I'm trying to make here is that it's just as reasonable for two implementations to exist concurrently in the same library, while otherwise being separate.

Related to this idea of implementations is that of *versions*: For example, in the above narrative the first two modules (following *judgment*) are that of *function* and *equality*. In practice these modules would actually only have the bare minimum content needed to offer dependencies for the modules following after. Why? It's tempting to try and define everything about these modules at once, but for these two (philosophical) concepts in particular, I would make the claim that there is enough evidence within the landscape of mathematical literature to say they are likely

[24] I have put in much research and development effort figuring out how to implement type theory from first principles. With that being said, any specific narrative is dependent upon its implementation language, and so based on that one's own type theory narrative might differ from this. I will claim though that even when such narratives do differ, they will likely only be of small variations.

[25] These specific keywords come from the scheme dialect of the LISP programming language. As such, I am potentially overgeneralizing, but I don't consider it controversial to claim these operators (or some variant in their names) are canonical across all such languages, and possibly even all functional languages.

[26] This type is more commonly known as a *coproduct*.

the most complex ideas mathematics has to offer. In turn, any module able to successfully express these ideas in full would likely come much later in the library as a whole. Given this, the idea of different versions becomes quite natural.

Finally, in terms of design extensions, I would like to discuss just how this schema's straticum and patronum could be used in actual programming languages, since it might not be immediately obvious.

For me, the first answer that comes to mind is that these models prevent the possibility of circular definitions. This is to say (and it takes us back to the methodology section), these models exist to mitigate language consistency issues: Everything has already been constructed (and thus defined), and although the narrative within the modulum might at times give the appearance of circular definitions, they're only navigational illusions.

With this being said, when observing actual languages such as C++ this problem is already solved in another way using what are called *declarations*: One declares a structure or variable without having to define or initialize it there and then. Given this, if current languages have solved this issue in their own way, do we really need a straticum and patronum?

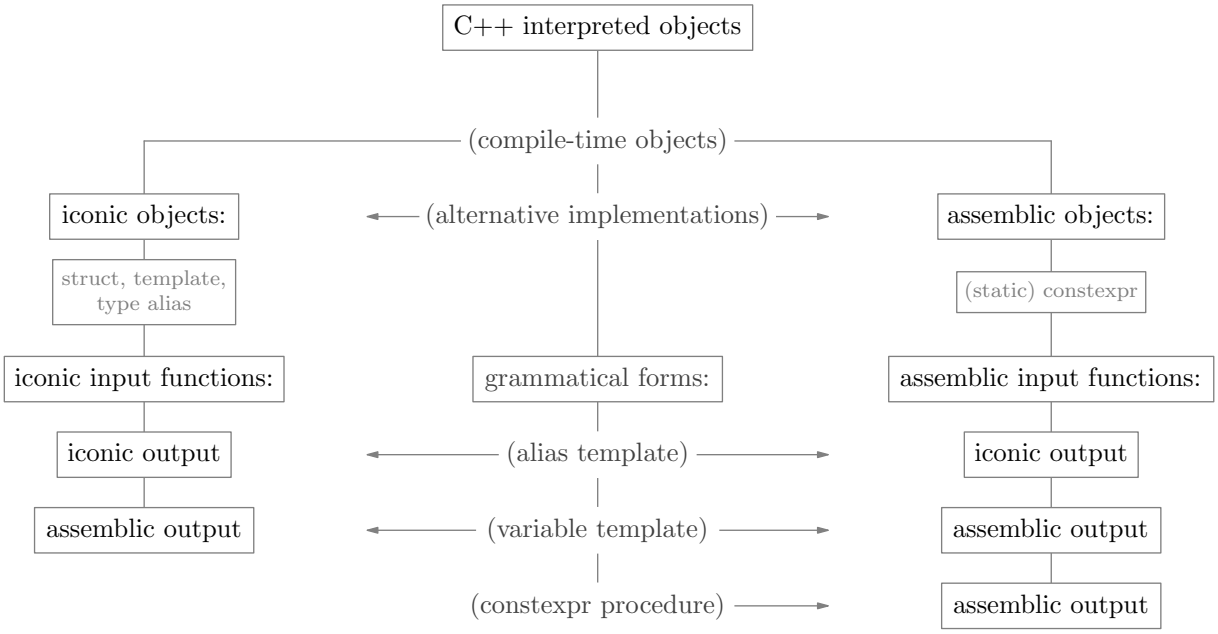
In this case I would say the greatest benefit of these models are in their ability to minimize object redundancy. In contrast to this, the nature of the modulum (or modules in other designs) is that we can actually have the same object or function more than once—being in more than one module. The point is, whether it's double counting across modules or by some other means, if a compiler is not aware of the issue it will effectively duplicate the same code within the same program, leading to what's known as *code bloat*. For example this is a known problem with C++ template programming.

If instead we were to take appropriate tie-breaking strategies to prevent straticum and patronum duplicates, a compiler supporting these models could then take an object or expression and *pattern match* it to figure out where it uniquely belonged. As actual programming languages are necessarily lazy, the compiler would check if said object or expression already existed. If so, it would only update names and locations related to the currently parsed module, and if not, then the object would be created for the first time. In this way, a compiler would then package together the required objects and algorithms translated from a given source code only once.

## C++

We've reached the final subsection. I have chosen here to discuss a single programming language example before we conclude this essay, and I have chosen C++.

As it turns out, due to what's known as *template metaprogramming*, C++ is actually able to support the stratica schema. For the most part it follows the existing programming language designs discussed above, but given that metaprogramming is technically "a hack", it is worthwhile to add one additional classification scheme to clarify its interpreted objects:



The original compile-time objects and functions of C++ were based off of *templated structs* and *integer enums*. It was determined that existing policies built into the design of the compiler actually allowed for instruction branching and evaluation such that this language "hack" was realized to be Turing complete. Since those early days meta-programming has evolved and been extended to include additional features such as *alias types*, *constexpr values*, *constexpr functions*, the latter two having the potential to be used both as compile-time and run-time objects.

This is where the above figure comes in: Building on the organizational designs of the previous section means we already have a rich scheme of overlapping classifications for our modelling objects. Now, within the scope of C++ meta we have even more object classifications, and they also overlap. This makes the process of interpretation much more complicated, and without a clear design for how these objects and grammatical forms relate to each other, narrative design becomes—and I say this from experience—way too confusing and disorganized!

In particular then, the above figure says that the two main kinds of C++ compile-time objects are **iconic** and **assemblic**.[27] The major difference between these objects comes from informally casting them as *template parameter* types. Iconic objects would cast as *type* template parameters, and even *template* template parameters, while assemblic objects would cast as *non-type* template parameters. On a casual level, it could be said that iconic objects are better suited to be *abstract*, while assemblic better fit the idea of *resolved*, but don't expect this description to always help clarify the issue.

In anycase, one other thing the above figure makes clear is that within the larger modulum narrative, we should expect more than one implementation of any given C++ module. For example if you implement a *judgment* module as in the previous section, it becomes quite reasonable to implement judgment objects as both iconic and assemblic. What's more, in such a case we would also potentially have five different implementations for any given function that takes judgements as input. This can be tedious, and in general there are various ways to mitigate the problem, but given the subtle nature of C++ meta it generally scales better to take the direct approach.

As for this subtle nature of C++, although it's not directly related to the whole of this essay it's worth mentioning the basics as they inform the stratica schema designs applied here. To put it succinctly, the nature of C++ meta-programming is that it is a best practice to always minimize *memoizations*.

The word "memoization" is a paradigm of programming where a function is implemented such that the first time it is called on a given value it calculates the output, and then stores that output in a specific memory. Then, every time that same function is called with that same value it only needs look up the output as a reference. The word is applied in the context of C++ meta because the nature of the C++ compiler when translating source code into machine code is to take any declared *struct* it comes across and do some prep work (such as resource allocation) on the expectation that it will be used to instantiate a run-time object. In a very general sense, this process resembles the memoization paradigm, hence the name.

The reason minimal memoization is a best practice is because objects which are strictly compile-time will never actually be used during run-time, so that the memoization work the compiler does becomes wasted cycles, which in practice only adds orders of magnitude to interpretation time—enough that it's worth putting in the extra effort to program in this style. To take this best practice further, we end up up privileging *alias templates* over templated structs when we can, not to mention coding meta-functions in *continuation passing* style.

As for how this memoization paradigm informs the stratica schema here: It is worth noting that the full type system of C++ is actually rather involved in its technical specification, and although its tempting to broaden the above classification scheme to include a wider range of compile-time objects, it is due to the above mentioned compile-time constraints that I have by design kept the use of distinct grammatical forms to a minimum.

I would like to finish here by saying this classification scheme is also a result of previous attempts to organize my own C++ *nik* library. In fact this library was one of the two major motivators in designing the stratica schema itself. The hardest part in its design for me was not the algorithms or the meta-programming constraints, but the classification of C++ objects to fit a clean narrative!

[27] This terminology is my own. I found it important to have umbrella terms for C++ grammatical forms considered related (within my library), not to mention the need for distinct names which wouldn't be confused with existing C++ nomenclature.

# Conclusion

We've now reached the end of this essay.

There have been a lot of individual designs, paradigms, schemes throughout to say the least, and as such I've been on the fence as to whether or not I should try to summarize them here. I have decided against it, and that it makes more sense to discuss two open problems instead regarding the schema as a whole.

The first is a theoretical concern: I relied heavily on [2] as the source for abstracting out and reusing the stratica pattern, and although I have proven the design offers at least a partial model for Set Theory, I have not proven it applies to all of Set Theory, and that's potentially worrisome.

The second is of practical concern: Within the stratica schema no specific grammar for pattern abstraction is given. In practice, if we were to use the straticum and patronum as a memory system with unique representation, there is no actual proof of decidability when it comes to translating modulum expressions back into their originals. This is to say: We might be able to classify objects in theory, but we will need actual algorithms if we want to automate.

Beyond that, I suppose there are other open questions, but the above two seem to be the most relevant currently. In anycase, as they have now been stated I would like to mention a personal note regarding the second of the two main motivators behind this essay (the first mentioned in the previous section, it being my C++ nik library).

I have been researching a foundational language comparable to Set, Type, Category Theories which I call *Concept Theory*. Without going into its details, we can think of concepts as providing a language for specifications. In application to this schema, the grammar of concepts are expressive enough to represent objects from all three of its models. This then admits the possibility of a mathematically potent programming language equipped with an organizational design (and a head start on a narrative design) that allows it to be both a proof assistant and a compiler. Although there is much work yet to be done, I hope for it to initiate a new genre of programming language design, for which this essay then is a major step in that direction.

To end, I would like to say thanks to the reader: Regardless of my personal intentions, I hope this essay was worthy of your interests and time, and maybe even of your designs.

# References


[1] H. Abelson, G.J. Sussman. Structure and Interpretation of Computer Programs (second edition). The Massachusetts Institute of Technology Press (1996).

[2] D. Nikpayuk. A Computational Model of Reference by means of Stratified Powersets (Part One) (2014). https://github.com/Daniel-Nikpayuk/Mathematics/blob/main/Articles/A%20Computational%20Model%20of%20Reference/Stratified%20Powerset%20(Part%20One)/stratified%20powerset%20(part%20one).pdf

[3] D. Nikpayuk. Grammatical Elements of Function Induction (2020). https://github.com/Daniel-Nikpayuk/Mathematics/blob/main/Essays/Function%20Induction/Version-Two/induction.pdf

[4] D. Nikpayuk. 1-cycle List Induction (2020). https://github.com/Daniel-Nikpayuk/Mathematics/blob/main/Essays/List%20Induction/Version-One/induction.pdf