A Computational Model of Reference by means of Stratified Namespaces (Part Three)

Daniel Nikpayuk

November 20, 2017

This article is licensed under
Creative Commons Attribution-NonCommercial 4.0 International.

Abstract

In part three we take the data structure resulting from part two and explore its properties to derive a model of language translation to be used within the compilers of a new genre of programming languages.

The translation process itself will be decomposed into a stratification of namespaces, each layer focused on translating names with ever increasing complexity as we move away from the target architecture. By refining our understanding of declarative names in particular, we will show this partitioning of layers to form a hierarchy of data, structure, concept, and type.

Following this we will finish with a quick tour of the advantages such a translation process offers, along with an example of how one could implement a linked list in principle.

Review

In part two we introduced a *weak specification*, what was called a *midway*, or *intersection* between (strong) specification and implementation. In particular this weak data structure was and can be decomposed by its *definition*, as well as its *operators*.

Visually, the data structure—which we called a *stratituple*—is as follows:

	$\mathbb{T}^0$	$\mathbb{T}^1$	$\mathbb{T}^2$	$\cdots$
$\mathbb{V}^0$	$s_0^{(0,0)}, s_1^{(0,0)}, s_2^{(0,0)}, \ldots$	$s_0^{(0,1)}, s_1^{(0,1)}, s_2^{(0,1)}, \ldots$	$s_0^{(0,2)}, s_1^{(0,2)}, s_2^{(0,2)}, \ldots$	$\cdots$
$\mathbb{V}^1$		$s_0^{(1,1)}, s_1^{(1,1)}, s_2^{(1,1)}, \ldots$	$s_0^{(1,2)}, s_1^{(1,2)}, s_2^{(1,2)}, \ldots$	$\cdots$
$\mathbb{V}^2$		$s_0^{(2,1)}, s_1^{(2,1)}, s_2^{(2,1)}, \ldots$	$s_0^{(2,2)}, s_1^{(2,2)}, s_2^{(2,2)}, \ldots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Our rows are called *unified powertuples*, the cells that make up these rows are called *powertuples*, [1] and the names within cells are called *sifter names*.

As for the lifecycle operators, we have:

- **constructors**: These operators take care of the allocation and initialization of this table structure. Effectively we have a list of lists of lists (3-dimensional listplot) of names. Each cell is initialized to be empty.
- **destructors**: These operators take care of the deallocation of this table structure. We need to safely destroy sifter names within cells, then the cells themselves, then the rows.
- **extenders**: These operators accept requests to add new sequences of names to the table. The general approach is for them to read the input sequence, determine to which cell it belongs, validate the names that make up the sequence (make sure they already belong to the table), as well as verify that the sequence itself *doesn't* already belong. In which case the new sequence would be added, and given a name.
- **restricters**: We can delete a sequence and its name from a given cell if no other sequence depends on that name in its own extension.
- **mutators**: We can mutate a name's sequence in a given cell if no other sequence depends on that name in its own extension.

Keep in mind, both here and in part two we have only dicussed *write* operators. This is in contrast to *read* operators—ones which read the names and sequences within our stratituple table. Discussion of such operators is omitted as the simplest of them are trivial, while the complicated ones are otherwise application specific.

Applications?

So far, in the previous articles and in the review we have introduced a data structure which has the potential to represent all other data structures. We've reached the point where it's time to ask the all important question: What are the applications of this stratituple?

extensible markup language

The first application I will mention which was actually one of the original motivations for this data structure is that of XML. For those who might not know, XML is short for *extensible markup language*. XML is a specification allowing one to mark up text in a way similar to HTML. The idea is it gives you enough freedom to create your own markup tags, but otherwise you have to adhere to certain grammatical notation to structure your text and express your tags.

I myself was introduced to XML while exploring the *digital humanities*. Humanists have been interpreting and marking up literary texts for centuries. It is a recent trend that they have turned to XML to standardize their critical, hermeneutic, and other analyses, offering new automated ways to cross-reference and cross-check against their analytical models, as well as in general find new patterns in text not previously seen—by human eyes alone.

As part of their adoption of this technology these digital academics, in a very humanist way I might add, have deconstructed and problematized the XML technology stack. In particular they adhere to demonstrating that one of its biggest limitations for their purposes is the redundancy of content. This limitation in particular stems from the fact that tags in XML are structured as trees, or in theoretical computing science terms: "context-free grammars"—meaning they are hierarchically structured with no overlapping scopes allowed.

This is a problem, because in the application of interpreting text, grouping and regrouping parts of a text is an all too natural action taken by these researchers. Humanists study complexity and take it just as seriously as information theorists, mathematical biologists, etc.—their subject of study being the range

[1] In the case we need to clarify, we could also refer to these cells as *regular powertuples*.

of human experience expressed within the world's literature after all. Overlap of textual interpretation is a fact of life.

In any case, having reached a limitation in XML's ability to represent complexity, one of Digital Humanities' most frequent workarounds is to modularize these marked up interpretations such that any given module has no overlapping scopes. As a consequence, one would have to make separate copies of the original text so as to create these individual non-overlapping marked up modes. This is an effective solution of course, but it creates an unnecessary duplication of the original content as well as possibly separating and blurring intratextual relationships: It's not optimized for approaches such as comparative analyses as it becomes more difficult to find connections between independent modules.

As to why our stratituple might be a good application to this particular problem of textual markup: If you consider the original cell in a stratituple as the unmarked text, then every tree constructed to structure this content is equivalent to an XML markup tag. Except here, a stratituple is optimized to allow for overlapping scope. Another way of thinking about a stratituple then would be to take an XML syntax tree and refactor its markup reducing redundancy and improving clarity. Overlap is allowed, nothing is wasted.

Keep in mind a stratituple wouldn't be able to simulate XML with perfect ease if we had such a goal in mind, but that's not the point here: It's not meant to be a perfect replacement, it's meant to offer an alternative to this specification altogether.

compilers and interpreters

Our main application of interest—and the core topic of this article—is in using our universal data structure as a model of memory reference for a new genre of programming languages. In particular our stratituple can be used within executable code itself, or at the very least as an engine of translation within the compilers and interpreters of these languages.

To get a feel for how this would work, it might be worthwhile to first take an inventory of the strengths and potential features expressed within the design of this data structure.

redundancy minimization

We've already alluded to the first feature of our stratituple when we discussed the limitations of XML: In order to effectively achieve overlap of scope we must create unnecessary memory redundancy within this markup specification as "data structure". This is caused by the creation of separate copies of the original text while structuring the independent marked up documents.

In the first article, I brought up the example of *a pair of pairs* (as data structure) being implemented in set theory:

$$((a,b),(c,d)) \quad := \quad \{\{a,\{a,b\}\},\{\{a,\{a,b\}\},\{c,\{c,d\}\}\}\}$$

As was noted, this was and is too confusing as a practical definition: There are too many left/right curly braces, there are too many repeated uses of the symbols 'a', 'b' , 'c', 'd' on the right hand side. Using this as an example, how could we reinterpret this within the context of our stratituple? Given the redundancy in our set-theoretic definition, we are inspired here to modularize out the original content from its structure. In the above example, such a modularization would be along the lines:

$$\{a,b,c,d\} \quad , \quad \langle\langle\cdot,\cdot\rangle,\langle\cdot,\cdot\rangle\rangle$$

which is much more accessible: We can now see the content as well as what the structure is suppose to be. Admittedly, it's currently less clear how the content maps directly to the structure itself, but more will be said on this later.

It's vague at this point, but this principle is called *redundancy minimization*, and is a core part of our intended design. [2]

[2] Beyond this particular motivation, redundancy minimization in the application of code production of higher constructs, paradigms, design mitigations (modularity, extensibility, etc.), comes not only from the need for readability, clarity, code maintainence, but just as importantly for the need to reduce *redundancy* for its own sake. Every redundant element is a chance to introduce a bug. Aside from the ever pleasant labour saving costs of reducing the number of times you have to repeat yourself, minimization also makes code validation easier as there's less to look at, and so less relationships to keep track of. The point is in contexts such as these it can be argued compressing or even minimizing redundancy is a public good.

navigational orthogonalization

Our modularization of the original content

$$\{a, b, c, d\} \quad , \quad \langle\langle\cdot,\cdot\rangle,\langle\cdot,\cdot\rangle\rangle$$

in our set theoretic example above brings up the second main design feature: navigational orthogonalization. Modularizing content from structure not only saves memory, it in fact parallels our philosophical considerations in the first article of a set with no inherent structure paired with an identity data structure to access its elements. In viewing it this way, we can from the initial design provide "native support" for the idea of navigational spaces as low cost records composed of (and conceptualized) as *sifters*: names with structural and navigational content embedded within.

media manipulation

Building upon navigational orthogonalization where our (composite) sifters act as lightweight records of how we've structured some given content, we are in a position to shape and reshape the structure surrounding that content in a low cost way.

An example would help, but for the sake of clarity and simplicity let's work with sifters which directly show the content they point to:

$$\langle \text{m, a, r, y, a, m, \_, m, i, r, z, a, k, h, a, n, i} \rangle$$

is a sequence pointing to character content, but we could also take this content and reshape it as a wordlist:

$$\begin{array}{l} \langle \\ \quad \langle \text{m, a, r, y, a, m} \rangle, \\ \quad \langle \text{\_} \rangle, \\ \quad \langle \text{m, i, r, z, a, k, h, a, n, i} \rangle \\ \rangle \end{array}$$

Note that these two objects only differ in their composite sifter structures, not in their content.

It is a design feature of a stratituple that it allows for expressivity of structural translation. The modularization of content from structure given its lightweight representation serves well in prototyping a variety of structures and could even be used for multimedia conversion.

This is not to say shaping and reshaping structure has no computational cost at all, but think of it the way we maintain and manipulate *filesystems* for example. A file might take up a large share of the secondary storage (memory) its held on, but the pathname, filename, permissions, etc. are much smaller and more accessible to work with when you're only looking to navigate or organize the existing filesystem.

We take for granted the fact that if we move the file into another location in the filesystem we're not moving the entire memory of the file itself on the hard drive, which could be quite large in size and expensive to move for all we know, we're moving a symbolic representation of that file on the filesystem alone. In this case, the filename is the primitive sifter, and the path of its location is its composite sifter as a record within the larger filesystem data structure.

We leave the content in place and manipulate it symbolically with its handle.

type reinterpretation

Although this stratituple has an implicit type system for classifying its sequences (mentioned in *part two*), these composite tree structures otherwise are collections of names and thus have no inferred types on their own. If we accept such ambiguity at this level, we then are offered an opportunity to modularize the type system away from our content.

Every modern programming language has a builtin type system. This is important for finding errors in the source code at compile-time so they're not introduced as bugs during run-time. The limitation of these systems is that a fixed type is a fixed type: Builtin types and any composite types constructed from them can't be changed. This limits the user's ability to reinterpret a type for a new situation, which in the long run lowers design entropy as it constrains ones ability to reuse code. As this is a known problem, the general workaround is to in some form or another be able to "alias" a given type so you can rename it, but this is a superficial solution as it does not play to the strength of human innovation: The ability to look at something and see it or repurpose it in an entirely new way.

If we allow for the modularization of types away from their underlying untyped structures, to offer types as *first class citizens*, then the interpretation and reinterpretation of the underlying tree structures (that make up this stratituple) become possible, and become the doorway to reusing these structures in an exponentially greater variety of situations.

A new genre of programming languages

With the large diversity of existing programming languages already out there why create another? In particular, how will this genre of programming languages differ from **procedural** languages such as C++, or **functional** languages such as Haskell?

How do languages in general differ from each other in the first place? It might be a bold claim to say I offer another genre, especially since as of yet at the time of this writing no such actual language even exists! Languages differ in their grammar as well as problem solving focus (which paradigms are easier to code for example), but they also differ in their model of how they translate source code into machine code. With a current lack of detail in the former, I will now discuss the latter.

The idea here in this model of code translation is that we break up the expected translation path into simpler paths, where each adjacent path lightens the load by translating smaller differences than if one were to do it all at once. [3] The unifying narrative informing this decomposition is the idea of a *name*, and in particular that of a declarative name.

The strategy of translation I offer is to realize the majority of meaningful expressions in any given computational language are descriptive in nature. Hence, if we interpret a given source code in terms of its names, and then stratify those names into namespace layers of increasing complexity, we can translate accordingly. As to how this model relates to languages like C++ and Haskell? If I were to offer a single sentence answer: You can think of the lower layers as possessing features more procedural, with the higher layers being more functional.

The linchpin in all of this of course rests on the idea of being able to distinguish complexity within names. This begs the question: Are names really that complicated?

What's in a name?

In order to manipulate something algorithmically or in code we would first need to describe *what* it is we're computing, and then *how* we're computing it. This is to say the value of a name in a computational language is in its ability to *denote*, or *refer*.

To that end, what's in a denotational name? What does it mean to reference?

We start with the obvious, *direct references*:

$$x \to 2$$

Here we have a name x and it maps to the number 2. This is an example of the simplest and most well known variety of referential name. We will call this a *primitive direct name*. This is to contrast with the idea of a *composite direct name*, which is where we take primitive direct names and combine them, for example:

$$(x, y) \qquad \text{where} \quad x \to 2, \quad y \to 5$$

Is this really a name though? Or is it just two names in disguise? Of course it's a name! One could for clarity more directly map it as follows:

$$(x, y) \to (2, 5)$$

[3] Such a translation approach suffers a trade off of course: More piecewise paths mean less work to be done in each—which offers a more refined translation, but more paths also means more chances for the message to get "lost in translation" overall. It becomes a matter of finding the optimal number of paths as well as good choices of adjacent paths to translate.

and so you can consider the whole string "(x, y)" as a single name. Another example is: "Ada Lovelace", which is composed of simpler names (letters and a space)—each themselves referring to other things—and yet we would never question its legitimacy as a name.

Since we're extending names, one might ask if we can take direct names and combine them in ways other than through pairs or strings? Technically yes, but as our stratituple is based on finite length sequences, the only compositions worth looking at within this theoretical framework are those of names with sequences as their underlying combining structure. [4]

Okay, so is that it? Direct names? Simple and composite? . . . not quite.

For our purposes, we need a denotational framework that is sufficiently powerful and expressive, and direct names alone just don't cut it. The easiest way to put it is we need to weaken the way in which we use names as references.

The next level of complexity in our naming system comes from weakening our names by *deferral*: This is where we allow for a delay with a promise to refer later on—a *deferred referral*, if you will. We do this by creating a dependency; we do this by letting a name refer to a possible range of elements from a larger context, a *set* of objects, while withholding until necessary exactly which one.

There is of course more than one way to achieve this effect, and there are two particular interpretations worth looking at: *scope* and *potential*. To defer by scope is to add the logical quantifier: "there exists" ($\exists$). To defer by potential is to add the logical quantifier: "for all" ($\forall$).

Let x be our name again, and S its context. Deferring by scope works by saying "there exists an element of S to which x refers." Thus we still have a referential name, it still refers to a single object at any given time, but now we don't specify exactly which object it refers to—only a range of possible objects for which it does. The assumption is at some point we will know to which specific object it does refer.

As for deferring by potential, it works by saying "x can refer to any element of S". One could also view this as a name which preserves relationships. A true placeholder. It is the act of recognizing and refactoring a pattern, holding the place within a set of relationships at the location where the unfactorable variation occurs. Such a name defers to the potential—the context of variants for which it *can* refer.

Deferral by set is powerful on its own, but we can achieve a greater flexibility if we take deferral by *subset* into consideration. The common approach within set theory when subsetting is by *predicate*, which is a natural fit here anyway as we've already introduced the logical quantifiers $\forall, \exists$. [5] Let's look at a few examples within set theory to get an idea of how we'd go about this.

declarative names

We start by declaring the definition of an *inductive set*:

$$\begin{array}{llll} \text{Let} \quad \mathbb{I} \quad \text{refer from type } \textbf{set} & \text{such that} & 0 \in \mathbb{I} & \\ & \text{and} & \forall x, \quad x \in \mathbb{I} \quad \Longrightarrow & s(x) \in \mathbb{I} \end{array}$$

$$\begin{array}{rrcll} \text{where} & s(x) & := & x \cup \{x\} & \quad \text{(the successor operator)} \\ \text{and} & 0 & := & \emptyset & \end{array}$$

The intersection of all such inductive sets is generally considered the definition the natural numbers from a set theoretic construction. [6]

So let's break down what we did. We started with the keyword "Let". This right here alone is a lot to unpack as we are making a declaration. When we *declare*, it might not be apparent at first, but this is only subtly different than deferring a name by scope. Both refer to a range of possible objects, but a declaration has the *intent* of being descriptive, while a scope deferred name has one of being dependent.

Following our declarative keyword, we then specify our "$\mathbb{I}$", our name.

Next we specify the *type*, it specifically being a **set**. Type theory itself is beyond the scope of this article, but we use ideas from it here: Notably, when the context is limited, a modern interpretation of a type is

[4] The additional value of limiting ourselves to finite sequences is in the need to prove the uniqueness of a name—so that our combined name (x, y) for example only maps to one unique value pair. This is possible because the identity of a sequence is dependent only on its componentwise identities.

[5] This of course was no coincidence.

[6] Although this is how one constructs the natural numbers, keep in mind it is an axiom of set theory, an assumption, that such an inductive set even exists in the first place.

as a set, where type instances are then the elements. As such, it will be taken for granted that we can interchange these terms without issue. With that said, it would be remiss of me to suggest that types and sets are identical, as it is known you cannot have *the set of all sets*, for example. [7]

Finally, after the type specification we mark the introduction of our predicate with the keyphrase "such that" indicating the logical statement following is what narrows down the scope of our denotational name to something smaller than the original type. This is to say we *subtype.*

Let's try another example from calculus, that of continuous functions ($f : D \to \mathbb{R}$):

$$\text{Let} \quad \mathcal{C}(D) \quad \text{refer from type } \textbf{set} \quad \text{such that} \quad \forall f \in \mathcal{C}(D), \quad \forall U \text{ open in } \mathbb{R} \quad \Longrightarrow \quad f^{-1}(U) \text{ is open in } D$$

or if that's not your interest, we could declare the cartesian product of natural numbers:

$$\begin{array}{llll} \text{Let} \quad \mathbb{N} \times \mathbb{N} \quad \text{refer from type } \textbf{set} & \text{such that} \quad \forall a, b, \quad a, b \in \mathbb{N} & \Longrightarrow & (a, b) \in \mathbb{N} \times \mathbb{N} \\ & \text{and} \quad \forall x, \quad x \in \mathbb{N} \times \mathbb{N} & \Longrightarrow & \exists a, b \in \mathbb{N}, \quad x = (a, b) \end{array}$$

in regular math notation, we might express this example more succinctly as:

$$\mathbb{N} \times \mathbb{N} \quad := \quad \{ \ (a, b) \quad | \quad a \in \mathbb{N}, \ b \in \mathbb{N} \ \}$$

but as we're looking to understand the denotational name structure, it's better to express it in the longer form previous.

The common **pattern** within each of these examples could be stated as: *declare, name, type, predicate.*

This isn't the only pattern worth mentioning though: The first two declarations—the *inductive set*, the *continuous functions*—both use their predicates to describe general relationships within or between the objects of their respective sets. In comparison, the third declaration—the *cartesian product*—uses its predicate to describe the representation of the objects within its set. These ideas I would term as *conceptualization* and *abstraction.*

conceptual names

Conceptualization can be thought of as a formalization of *pattern recognition.* The simplest way of creating a *concept* is to divide an existing body of elements using some one-time-use *separator*, and then classifying one side accordingly. [8] This approach remains valid, but mathematicians have long known a more powerful strategy: *semantic filters*, which allow you to say things about a concept without regard to the original elements or their direct properties.

Take *continuous functions* for example. By declaring a semantic filter as we have done above, we are using a predicate in a novel way: We are not trying to divide elements to define a concept, rather the division mechanism itself is the concept. Furthermore, as a predicate is based on symbolic logic it means these semantic filters are also equipped with an algebra. The consequences of this being we can logically prove things about this concept independent of knowing any specific properties of individual elements satisfying its definition.

abstract names

Abstraction can be thought of as a special and limited (but important) case of conceptualization. Here we declare an abstraction as a concept, but its filter has a special form which is generally achieved with the following paradigm: "all objects that follow this [. . .] pattern are of this [. . .] type, and all such objects of this [. . .] type follow this [. . .] pattern". Abstraction can be thought of as a formalization not just of pattern recognition, but of *pattern matching.*

Given that abstraction is achieved by specifying a single common clearly defined pattern for all objects of its type, we say that it is *constructive.* This is in contrast with general conceptualization which is

[7] If you want to speak of *all* sets as a given context you would need to specify it as a broader type or class object, the philosophy of which is beyond the scope of this article. Potentially the easiest way to resolve this particular issue here is to avoid the use of types altogether, using sets alone, but doing so would constrain us from importing all that's wonderful about type theory, which would be especially careless given how integral type systems are to modern compilers.

[8] Sometimes this also means whittling down a concept using several separators: Dividing, then dividing again, as much as necessary until you end up with the desired class.

not necessarily constructive and otherwise defaults to being *intuitive*. [9] For example, with our continuous functions we wouldn't look for a single common clearly defined representational pattern shared by all, such an abstraction oriented definition would be too restrictive. Rather our weaker non-constructive semantic filter is valuable in that it potentially allows for many abstract classes of functions—all sharing this concept of continuity.

So what's in a name?

To summarize: You defer a denotational name by *scope*, and if you change its intent from being dependent to descriptive you get a *declaration* instead. Once you realize the *predicate* attached to the grammatical form of a declaration acts as a *semantic filter*, you now have access to *concepts* which you can now manipulate with math and logic.

That's what's in a name! . . . and why it was worth understanding in greater depth before heading on to our model of translation. With that said, it is finally time to introduce **the namespace layers**.

the data layer

Whenever you're trying to design a software program or application you start with raw data, or original content. You then build an interface of interactivity around it, or you access it to perform calculations, or you measure it in some way, but you always start with the data. The context for which everything else exists in the first place!

Identifying the axiomatic data is possibly the most difficult part in any production design, but a general rule-of-thumb is to strip away everything that seems like clutter and focus on the core content, asking: What really matters here? There's nothing beyond that, but to put us as code designers into the right frame of mind.

the structure layer

The structure layer consists of primitive and composite direct names. It is our constructivist layer, in which we abstract out the common structural pattern of finite length trees—for which our stratituple is the natural fit. As we've spent a lot of time on properties of stratituples already, there's not much more to cover here but for one interesting subtlety, which has a powerful consequence in application to our genre of languages.

Let's begin again with a previous example:

$$\begin{array}{l} \langle \\ \quad \langle \mathrm{m, a, r, y, a, m} \rangle, \\ \quad \langle \_ \rangle, \\ \quad \langle \mathrm{m, i, r, z, a, k, h, a, n, i} \rangle \\ \rangle \end{array}$$

Informally—for example purposes—we could say this tree fits somewhere in our stratituple, but as a sifter name, how do we navigate *its* structure? *its* internal names? *its* content? The answer is we start with the name of this tree, which we'll claim to be: *str*. To navigate *str* then, we simply navigate the locations (numerically) of its composite sequences: *str*/0 (="maryam"), or if we want to go further: *str*/0/2 (='r'). [10]

The subtlety worth introducing comes in realizing we can not only use stratituples to record and navigate data structures in this fashion, we can also use them to record and navigate code instructions. This is to say: *trees double as algorithmic code.*

This might not be clear at first, so I will offer here an example from the grammar of mathematical notation:

$$e \quad :\equiv \quad (x+1)(x^2-1)$$

We take for granted that any such expression can be considered as a sifter-like data structure—symbolic with navigational content embedded within. The natural question to ask is: How to navigate such a structure?

We start by taking its name e, and use it to referentially navigate substructures and content as follows:

[9] See Brouwerian constructivism.

[10] Interestingly, the notation we've chosen for the *navigational path* parallels the notation of some filesystem pathnames.

$$\begin{array}{lcl} e/\mu & = & (x+1) \\ e/\mu/\sigma & = & x \\ e/\mu/\sigma^2 & = & 1 \\ e/\mu^2 & = & (x^2-1) \\ e/\mu^2/\sigma & = & x^2 \\ e/\mu^2/\sigma^2 & = & -1 \end{array}$$

Okay, what did we do here? Think of the various suffixes:

$$\mu,\ \ \mu/\sigma,\ \ \mu/\sigma^2,\ \ \mu^2,\ \ \mu^2/\sigma,\ \ \mu^2/\sigma^2$$

in the above as navigational paths. [11] Our μ is shortform for multiplication '$*$'. The σ then is summation '$+$'. [12] In the broadest linguistic terms, you can think of e as an expression belonging to a grammar with verbs. In our case the verbs are "multiplication", "summation", and as verbs have valency—the number of arguments you can assign—they also have a natural (numerical) mechanism for referring to their respective arguments.

Thus, e/μ^2 is our shortform to mean:

"The second argument of the (root) multiplication of the expression e."

which corresponds to (x^2-1).

Hence, although not formally proven here, I hope you can agree that given any finite constructive grammar whose sentences can be decomposed into trees we can reinterpret said sentences as data structures, using their verbs to navigate their internals.

The consequence of this within our language design is that we can—if we're careful about it—construct as well as manipulate algorithms themselves at higher levels. This extends our expressive ability to parallel the idea of *macros* that one finds in functional languages.

the concept layer

The concept layer allows us to give names to sifters in the structure layer. This is necessary as composite sifter names can become complicated and unwieldy in their own right. As one can roughly consider sifter names as the equivalent to data structures and/or content in other languages, the concept layer might be equivalant to adding variables. The difference here though is that these **filter** names—as was previously shown—can be used to act as concepts, deferring what they actually refer to until needed. How would this work though?

Within the declarative name paradigm discussed earlier, the general pattern was *declare, name, type, predicate.* If the concept layer is made up of declarations, then their common declaration type would be the stratituple, where instances of that type would be its sifter names. As such, any given semantic filter in the concept layer is formed through a combination of symbolic logic and properties specific to sifters themselves. As sifters are trees, any operator grammar that acts on filters will effectively describe traits and aspects of trees.

Building on this, a filter without any constraints defers by default to the whole stratituple as its scope. Any property we add to this filter allows us to narrow down that scope. As such, any simple (recursively computable) property of a tree can be used to pattern match for this purpose. The only condition is whatever collection of properties we maintain as our operator basis must form a filter algebra. Only then can we implement a concept theoretic system within the compilers and interpreters of this genre of language.

The one serious limitation to this way of using names is that any instance of a name deference needs to be resolved before an algorithm can act on the object it's meant to point to. A run-time program cannot continue while such an ambiguity remains; compile-time source cannot be translated while such an ambiguity remains. This brings us to the all important idea of *identity resolution.*

A necessary requirement source code must satsify within this genre of languages is that one should be able to modify any given filter sufficiently until its scope is restricted and it refers to exactly a single unique

[11] Or if you've heard of the internet before, as *urls*.

[12] This begs the question: Why not just use the star '$*$', '$+$' characters directly? It's less elegant to look at is all.

sifter in the structure layer. In that case, you can say the identity of the filter has been resolved. In effect, the idea is that our compiler/interpreter takes filters and will only translate them to the target architecture if their identities are known at the time of translation, otherwise an error would be gracefully reported, and compilation/interpretation would halt.

Finally, concepts have further value in that they can handle some of the burden normally carried by types. This concept layer offers among other things an alternative to type-theoretic *polymorphism*. With polymorphism, you constructively specify a *type* in your code, but you can leave part of that construction as a variable to be resolved later. With filter names on the other hand, you can represent structures and algorithms from a top down perspective, omitting target architectural details until the last minute—until they need to be resolved by the compiler for translation. You achieve the same effect, but with concepts you can access a certain expressivity of code you might not have with type theory alone.

the type layer

The type layer is all type theory, but within this framework it's more specifically about things like syntactic sugar and grammatical optimizations for applied contexts: creating modules (and then libraries) of types and their operators.

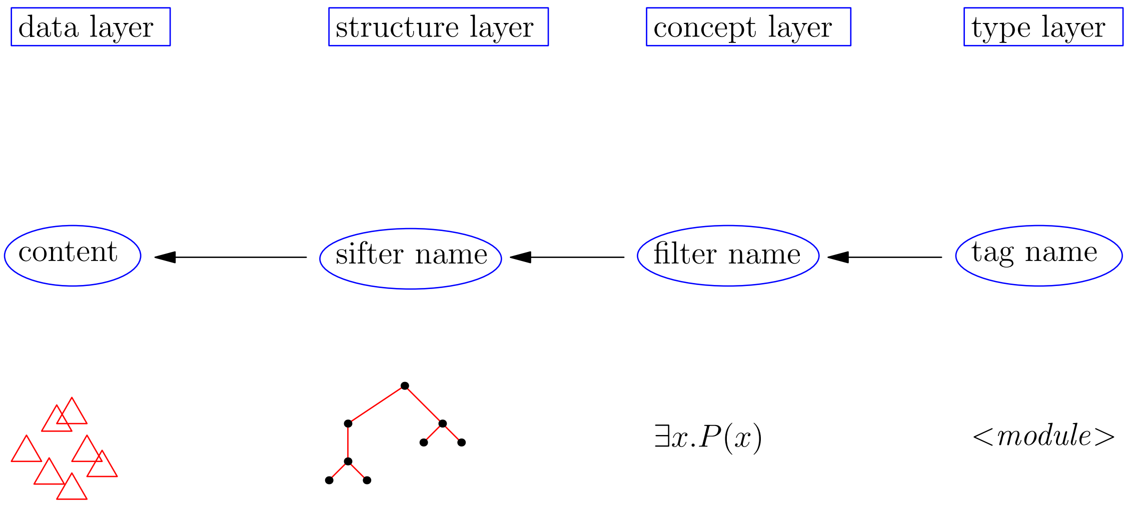
At higher levels like this source code becomes more about design theory, and how we organize our applications from more intuitive human perspectives. At these levels we create all the basic types coders have come to expect such as integers, characters, strings, as well as any required "real world" application types needed in production. [13]

It is here that *tag* names allow us to create and manipulate user defined types. This is similar to and inspired by template metaprogramming in C++. With user defined types, we can describe signifiers and cultural tags such as "first", "forward", "open", or "pushbutton" for example, or we can use them to create subtle and useful *dispatching* paradigms so that we can organize our code in new and effective ways.

I use the keyword *tag* intentionally, as it has connotations to XML and the way in which you can subjectively markup your code. Such tags allow us to mark filter names with additional meta content which only coders care about, and which compilers use during translation but which otherwise dissolve and disappear within run-time code.

Implementation

We summarize our stratified namespace as follows:



[13]It wasn't mentioned within the concept layer, but given that filters can substitute for polymorphic types, it is not unreasonable to create "object" modules at this level. As a result, weaker data structures such as lists, vectors, sets, maps, etc. would show up here.

translation efficiencies

I would like point out one of the efficiency advantages to this model of translation.

Within the type layer, I had mentioned template metaprogramming in C++ as the inspiration for user defined types. The limitation within C++ though is what's called "code bloat". In effect what happens is you implement two different user defined types which resolve to the same target architectural type but the C++ language and its compiler don't know this, so in turn duplicate copies of the same algorithm are frequently made. Of course users of the language have discovered many best practices to mitigate this known problem, but because the language was never designed with types as first class citizens from its inception, it will never be able to overcome this design inefficiency altogether.

The advantage with the stratified namespace model is that any two tagged names within the type layer eventually map through the structure layer where there is minimal redundancy and no distinction by type. The only types which resolve are the ones which map directly to target architecture types, the rest dissolve during compilation. This is to say: Code bloat never becomes an issue.

Furthermore, by inverting the level at which types are introduced within the language, I make the claim it promotes cleaner narratives and better design thinking. For example, within C the builtin primitive objects are already typed, such as the well known *int*. There's clear reason to package together this object type along with its operators (registers possess numeric hardware subroutines), but it inadvertently promotes less effective designs as the types of these builtin names (numbers) can't be changed.

You should be able to take these numerics, these numeral strings and interpret and reinterpret their types (and thus what operators they come grammatically equipped with) as you like. Optimization isn't even a problem, for if you specify an integer type and thus equip these names with numeric operators, you can easily translate to the target architecture and call the hardware subroutines directly. You still get the best of both worlds.

linked lists

Before we conclude, I wanted to introduce how we might implement a linked list within this genre of languages. As no actual language grammar has been codified, we will be relying on something like pseudocode, vying to present the core ideas instead of the grammar.

In a low level (access) language such as C++ a simple implementation of a linked list would be built from a node, itself defined as an object-oriented class. Such a node would contain as members a *value type* and a *pointer* which stores the location of the "next" node. Everything else such as the linked list's operators are extended upon that. As such, I will not offer the full implementation here, just enough of the node implementation to get the point across—focusing on how to access these nodes and how to find their values or their next nodes.

To implement a node in this genre of language, we would start at the structure layer. A node needs a value as well as a pointer to another node:

$$\text{node1} \quad := \quad \langle \text{value101},\ \text{node2} \rangle$$

Keep in mind any given name within a stratituple can be conceived of as a triple:

$$\text{value101} \quad := \quad \langle \text{row198},\ \text{column317},\ \text{name12} \rangle$$

but in the case of node names themselves, we can optimize: If we knew in advance that all node names occurred in the same cell, then when referring to node names we'd only need the name within the cell and not the row/column names (which we would know to be fixed). Given that a node is a pair—consisting of a triple of numerals as well as a direct numeral—it follows that such a node would reside in the $(1, 1)$ cell of the stratituple.

As for the above names such as "row198" or "node2", strictly speaking these sifter names would each represent untyped names (effectively just address locations) which would be interpreted and reinterpreted as needed, but for clarity here I have embedded type content within the names themselves. I hope that's not too confusing, but if you've worked with pointers before, you know how they can get.

In practice, it would be more advantageous to use the grammar within the concept and type layers, as we would have access to more powerful grammatical operators. For example we might start by declaring a sifter name:

$$\text{sifter } \langle 1,1,5\rangle \quad = \quad \langle\langle 17,3,8\rangle, 103\rangle$$

but maybe we want to work with this at a concept level:

$$\text{filter } node \quad = \quad \langle 1,1,5\rangle$$

so that we could refer to its internals more readily:

$$node/0$$

but if that's still too obscure, we could work with it at an even higher level:

$$\text{tag } myNode \quad = \quad node$$

and then assuming we've tagged things accordingly, we could express things like:

$$myNode/value, \quad myNode/next$$

or we could take things further by defining and verifying concept logic, or we could pattern match, or add natural language tags for design organization, or add types to help us validate against bugs. In any case, I hope this offers a hint at the expressive power and general approach possible with these languages.

Conclusion

I will leave this article—the last of three in its series—by stating my own personal motivations in researching this genre of programming languages, as well as offering some final cautions for those interested in pursuing these ideas themselves.

Coming from a math background with a strength in manipulating formulas and identities (solving equations) I found an honest lack of expressivity for shaping and reshaping code and data when I took up programming. Furthermore, mathematics uses names in ways I think would be tremendously useful within code, but I have also found that lacking. Moreover, not only did I come from a math background I also spent much of my math education learning foundations, where one learns to see branches of math as perspectives which shape your thinking, which offer you *new ways of thinking.*

I have not found a programming language built on a theory of computing science which shapes your thinking in terms of what I've found in math. I would like to see languages that get you to ask what computation even is, or how design thinking is applied once you move beyond the basic grammar of a given language. There are many other wonderful languages which *can* teach these things, but I haven't found many which were designed to do so in the first place; that offer the narrative axiomatic approaches I have come to trust from mathematics.

With that said, if you were to pursue the narratives stated here, I would caution:

1. These languages are likely better suited as prototyping languages. Not to say they aren't elegant or efficient for most purposes, but if you're looking for that extra level of optimization, the stratituple layer might not be able to satisfy. An alternative outlook is to consider the model of translation presented here as a template for compiler *front ends.* The point of departure being the stratituple layer, where one could unpack its content within the target architecture according to your preferred metric of efficiency. The value as a front end is the expressivity and the existing layers of translation which would shed the excess of the source in preparation for the back end.

2. I have not designed this model of translation with safety or security in mind. As this article is only an outline for such languages, doing so is not out of the question. I only bring it up because at the time of this writing the real world of interconnected devices are overwhelmed with considerations of hacking, exploits, security leaks, etc. I am of the consensus that security has to be embedded at every level of a protocol, or technology stack—such as access privileges—and so one might wish to start here.

In any case, I hope the ways of thinking presented here may be of value to you, and regardless of the language you choose to code in: May your programs, applications, projects come to fruition, and may you succeed in all your future endeavours.

Pijariiqpunga.