A Computational Model of Reference by means of Stratified Powertuples (Part Two)

Daniel Nikpayuk

October 14, 2017

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

Abstract

In part two we take the mathematical model resulting from part one and translate it to derive a computing science specification for a universal data structure—a structure which can represent all other finite computable data structures.

Before this, we first introduce a variant mathematical model which is better optimized towards register machine architectures. The benefit in doing this is it allows for a natural translation of our mathematical model into the desired computatational specification.

Review

In part one we introduced our universe of discourse $\mathbb{U}^\infty$:

Stratified Powerset:

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

and we showed that this *low-level space* could be equipped with an algebra (a few rules of grammar satisfying closure within the space) modelling all the computable data structures we'd be interested in [1] :

Theorem 1 (Closure of Pair)

$$A, B \in \mathbb{U}^\infty \quad \Longrightarrow \quad \{A, B\} \in \mathbb{U}^\infty$$

Theorem 2 (Near Algebra of Sets) *Let $A, B \in \mathbb{U}^\infty$ with A, B as collectibles:*

1. $A \cup B \in \mathbb{U}^\infty$
2. $A \cap B \in \mathbb{U}^\infty$
3. $A - B \in \mathbb{U}^\infty$

[1] It was not discussed in part one, but our rules of grammar only allow us to construct finite data structures, leaving out those infinite structures accessible through the lazy evaluation paradigm. I would point out this may be a limitation of expressive potential, but not of data structure potential.

Stratified Powertuples

Before we discuss the proposed implementation, we need to soften the translation process by first reorienting our mathematical model around *sequences* instead of sets. As it turns out, this sequence based theory is equipotent to our set-theoretic orientation—in fact it almost identically parallels what was covered in *part one*—and so we won't need to discuss it at length here.

Our "stratified powertuple" is defined as follows:

Stratified Powertuple:

$$\begin{aligned} \mathbb{V}^{\infty}(\mathcal{S}) &:= \textstyle\bigcup_{n\geq 0} \mathbb{V}^{n}(\mathcal{S}), \quad \text{where} \\ \\ \mathbb{V}^{n}(\mathcal{S}) &:= \textstyle\bigcup_{k\geq 0} \mathbb{T}^{k}(\mathbb{V}^{n-1}(\mathcal{S})), \quad n \geq 1 \\ \mathbb{V}^{0}(\mathcal{S}) &:= \mathcal{S} \end{aligned}$$

Our notation needs names, so from now on we refer to $\mathcal{S}$ as our content set, while $\mathbb{T}^k$ are *powertuples*, and $\mathbb{V}^n$ then are *unified powertuples*. At this level we have effectively only substituted the letters $\mathbb{V}$ for $\mathbb{U}$ as well as $\mathbb{T}$ for $\mathbb{P}$ in our modified mathematical model. So if our use of $\mathbb{P}$ was to denote a powerset, what then is a *powertuple*? You can think of it as taking the elements of a set $\mathcal{S}$ and collecting them into all possible finite sequences of those elements:

$$\begin{aligned} \mathbb{T}(\mathcal{S}) &:= \textstyle\bigcup_{n\geq 0} \mathcal{S}^{n}, \quad \text{where} \\ \\ \mathcal{S}^{n} &:= \{\ \langle s_0, \ldots, s_{n-1} \rangle \mid s_k \in \mathcal{S},\ 0 \leq k < n\ \} \\ &\quad \text{that is, all sequences of elements of } \mathcal{S} \text{ of length } n \end{aligned}$$

From part one, many of the lemmas immediately translate, and as they do not rely on the definition of a *powertuple* their proofs are identical as well. We thus offer the following without proof:

Lemma 3 (Monotonicity)

$$0 \leq m \leq n \quad \Longrightarrow \quad \mathbb{V}^{m} \subseteq \mathbb{V}^{n}$$

Corollary 4 (Mobility)

$$0 \leq m \leq n, \qquad A \in \mathbb{V}^{m} \quad \Longrightarrow \quad A \in \mathbb{V}^{n}$$

Lemma 5 (Reduction)

$$A, B \in \mathbb{V}^{\infty} \quad \Longrightarrow \quad \exists m \geq 0, \quad A, B \in \mathbb{V}^{m}$$

Lemma 6 (Stratification)

$$\mathbb{V}^{n} \quad = \quad \mathbb{V}^{0} \cup \bigcup_{\substack{k\geq 1 \\ 0\leq m<n}} \mathbb{T}^{k}(\mathbb{V}^{m}), \quad n \geq 1$$

As was the case in part one, if the context is clear or no confusion will arise, we omit the content set $\mathcal{S}$. Moving forward, in regards to translation the definition of *collectible* has an immediate dual as well:

Sequentiable: Let $A \in \mathbb{V}^{\infty}$, with $A \neq \langle\rangle$. A is said to be *sequentiable* if

$$\exists j \geq 0, \text{ and } \exists k \geq 1, \quad \text{such that} \quad A \in \mathbb{T}^{k}(\mathbb{V}^{j})$$

here though the definition is subtly different than that of a collectible because our element A of interest cannot be the empty sequence $\langle\rangle$. Again, the intuitive idea behind a sequentiable is that it is an element which belongs to some powertuple of the whole construct. The value of this definition is it allows us to distinguish between content and structure. You might argue we could define a sequentiable then as those non-content set elements. Unfortunately we cannot assume that an element in the content set doesn't show up by chance elsewhere (as we do not in general know the nature of the content set). As such we will continue to rely on the weaker definition given above.

From here, we turn to the algebra of this data structure *space*. The idea behind the following *closure* theorems is inspired by that of the *lifecycle* of a data structure in real world machine processes. We will need to assure our ability to grow and shrink arbitrary data structures, both within a given level of complexity as well as across levels of complexity.

We start *across* levels of complexity, allowing us to build up *nested* sequences:

Theorem 7 (Capsulation)

$$a_0, \ldots, a_M \in \mathbb{V}^\infty \quad \Longrightarrow \quad \langle a_0, \ldots, a_M \rangle \in \mathbb{V}^\infty$$

Proof

$$\begin{aligned} a_0, \ldots, a_M \in \mathbb{V}^\infty \quad &\Longrightarrow \quad \exists m \geq 0, \quad a_0, \ldots, a_M \in \mathbb{V}^m \\ &\Longrightarrow \quad \langle a_0, \ldots, a_M \rangle \in \mathbb{T}(\mathbb{V}^m) \subseteq \mathbb{V}^{m+1} \subseteq \mathbb{V}^\infty \\ &\Longrightarrow \quad \langle a_0, \ldots, a_M \rangle \in \mathbb{V}^\infty \quad \blacksquare \end{aligned}$$

We also need to be able to break down nested sequences:

Theorem 8 (Decapsulation) *Let $A \in \mathbb{V}^\infty$ with A as sequentiable, then there exists $M, m \geq 0$ such that:*

$$A = \langle a_0, \ldots, a_M \rangle, \qquad \textit{with} \quad a_0, \ldots, a_M \in \mathbb{V}^m$$

Proof

$$\begin{aligned} A \text{ is sequentiable} \quad &\Longrightarrow \quad \exists m \geq 0, \quad \exists k \geq 1, \quad A \in \mathbb{T}^k(\mathbb{V}^m) \\ &\Longrightarrow \quad \exists m \geq 0, \quad \exists \ell \geq 0, \quad A \in \mathbb{T}(\mathbb{T}^\ell(\mathbb{V}^m)) \\ &\text{but} \quad \mathbb{T}(\mathcal{S}) := \textstyle\bigcup_{n \geq 0} S^n \\ &\Longrightarrow \quad \exists m, \ell, M \geq 0, \quad A \in [\mathbb{T}^\ell(\mathbb{V}^m)]^{M+1} \\ &\Longrightarrow \quad \exists m, \ell, M \geq 0, \quad A = \langle a_0, \ldots, a_M \rangle \\ &\text{with} \quad a_0, \ldots, a_M \in \mathbb{T}^\ell(\mathbb{V}^m) \subseteq \mathbb{V}^{m+1} \end{aligned}$$

In the case that $\ell = 0$ we have $a_0, \ldots, a_M \in \mathbb{V}^m$, otherwise $a_0, \ldots, a_M \in \mathbb{V}^{m+1}$. In either case this can be simplified to saying there exists $m \geq 0$. $\blacksquare$

Following this, we are ready to prove the elements of our stratified powertuple also grow and shrink *within* a given level of complexity while remaining in the space. In the context of sequences, this is naturally interpreted as a *lengthwise* property:

Theorem 9 (Catenation/Decatenation) *Let $A, B \in \mathbb{V}^\infty$ with A, B as sequentiables:*

1. $A|B \in \mathbb{V}^\infty$
2. $A \backslash B \in \mathbb{V}^\infty$

As we have not introduced the above notation previously, $A|B$ just means joining two sequences to become one long sequence, while $A \backslash B$ splits A returning the prefix of A that does not coincide with B. In the case B doesn't match as a suffix, A itself is returned. This operator can loosely be thought to parallel the *set difference* operator from set theory.

Proof

$$\begin{aligned} A, B \text{ are sequentiable} \quad &\Longrightarrow \quad \exists M, N, m, n \geq 0, \quad A = \langle a_0, \ldots, a_M \rangle, \quad B = \langle b_0, \ldots, b_N \rangle \\ &\text{with} \quad a_0, \ldots, a_M \in \mathbb{V}^m \quad \text{and} \quad b_0, \ldots, b_N \in \mathbb{V}^n \\ &\quad\ \text{without loss of generality let } m \leq n \\ &\Longrightarrow \quad a_0, \ldots, a_M, b_0, \ldots, b_N \in \mathbb{V}^n \\ &\Longrightarrow \quad \langle a_0, \ldots, a_M, b_0, \ldots, b_N \rangle \in \mathbb{V}^{n+1} \subseteq \mathbb{V}^\infty \quad \blacksquare \end{aligned}$$

Proof

$$\begin{aligned} A, B \text{ are sequentiable} \quad &\Longrightarrow \quad \exists M, N, m \geq 0, \quad A = \langle a_0, \ldots, a_M, b_0, \ldots, b_N \rangle, \quad B = \langle b_0, \ldots, b_N \rangle \\ &\text{with} \quad a_0, \ldots, a_M, b_0, \ldots, b_N \in \mathbb{V}^m \\ &\Longrightarrow \quad \langle a_0, \ldots, a_M \rangle \in \mathbb{V}^{m+1} \subseteq \mathbb{V}^\infty \quad \blacksquare \end{aligned}$$

These four operators though not computationally practical, or user-friendly as an interface grammar, are sufficiently powerful in their potential. Any implementation can easily extend these to more powerful versions optimized with intuitive grammars and acceptable efficiency standards.

With this we have now introduced our variant mathematical model. Before moving on to the implementation, a potential criticism does arise: If such a "stratified powertuple" model of reference is equal in potential, and is a better fit for translation, *why then was the set-theoretic version introduced in part one*?

It's not a bad question, but I've thought it over long enough I've decided it was worth keeping. A better question to ask to aid us in our understanding would be: *Let's say we had introduced the sequence-theoretic approach first (and only), what would we have lost in doing so*?

1. Set theory offers a foundation for all of math, allowing us a universal language to represent any mathematical or computational pattern we're able to recognize in our lives. There is no equivalent theory of sequences as a general theory of all mathematics. In the limited case of finite and computable, set theory and sequence theory can be considered equipotent because each construct can be used to represent the other, but without introducing the set-theoretic version first, we would potentially be restricting the theoretical limits of our theory as a whole.

2. Future computers such as quantum computers might not adhere to register machine architectures. Thus by introducing the set-theoretic version, we have introduced a version which is conceptually ideal, we are left with a template to which we can create any other variant model necessary when required. Powertuples are an optimization toward register machine architecture, but future architectures may differ, as such it is worth introducing this theory with as few structural constraints as possible, and set-theory offers that.

The value of introducing the set-theoretic version then is to maintain a higher level of abstraction and theoretical *portability*.

Implementation

Implementation comes in two parts: Our *definition*, then our *algebra*.

Definition

The implementation of our stratified powertuple definition can be summarized visually as follows:

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

This table is at the *intersection* between a data structure specification versus its implementation, of our mathematical model. You could consider it a psuedo description, but it's meant to be more refined than that: It's intended to be midway between any actual specification and any actual implementation. This is to say, it's abstract enough be refined to match any interface and policy specification, but it also has enough detail to offer a way of translating to any actual programming language or register machine architecture. This is its intention, but for the purposes of this article we will consider it on the side of a weak specification.

Before heading into it, we need a name for this data structure. As it's based off our stratified powertuple, for the remainder I will refer to this computing science style construct as a *stratituple*.

So let's explain how our stratituple works. The rows $\mathbb{V}^n$ can be thought of as the unified powertuples, and the columns $\mathbb{T}^k$ as the regular powertuples of our mathematical model. We'll call the locations where the rows and columns intersect *cells*. Looking back at the definition of a stratified powertuple, we constructively started with the content set $\mathcal{S}$. The elements of this set are translated into our stratituple as contents of its top left corner $\mathbb{V}^0\mathbb{T}^0$ cell:

$$s_0^{(0,0)}, s_1^{(0,0)}, s_2^{(0,0)}, \dots$$

We then construct each powertuple cell $\mathbb{V}^0\mathbb{T}^{k+1}$ iteratively from this basis. The second cell constructed in the column to the right of our initial cell is translated as the first powertuple of the elements in our initial cell. This is to say *only* [2] finite sequences of elements of the initial cell belong. So for example, our second cell might actually look like:

$$\langle s_0^{(0,0)}, s_3^{(0,0)} \rangle,$$

$$\langle s_1^{(0,0)}, s_2^{(0,0)}, s_5^{(0,0)} \rangle,$$

$$\langle s_3^{(0,0)} \rangle,$$

$$\langle s_2^{(0,0)}, s_4^{(0,0)}, s_2^{(0,0)} \rangle$$

This example does bring up a subtlety. In the main table our second cell is as follows:

$$s_0^{(0,1)}, s_1^{(0,1)}, s_2^{(0,1)}, \dots$$

so where did these names go?

[2] In the mathematical model we define a powertuple as *all* possible finite sequences of elements, but as we are dealing with an implementation with assumed limited resources we defer to a lazy constructive paradigm where we only construct what we use.

The difference is that the sequences shown above are the real contents of the cell, but for the sake of *ease of reference* we give them all simplified names:

$$\begin{aligned} s_0^{(0,1)} &: \langle s_0^{(0,0)}, s_3^{(0,0)} \rangle, \\ s_1^{(0,1)} &: \langle s_1^{(0,0)}, s_2^{(0,0)}, s_5^{(0,0)} \rangle, \\ s_2^{(0,1)} &: \langle s_3^{(0,0)} \rangle, \\ s_3^{(0,1)} &: \langle s_2^{(0,0)}, s_4^{(0,0)}, s_2^{(0,0)} \rangle \end{aligned}$$

To construct the next cell to the right (in the same row) then, we again construct sequences of names within the previous cell:

$$\begin{aligned} s_0^{(0,2)} &: \langle s_0^{(0,1)} \rangle, \\ s_1^{(0,2)} &: \langle s_8^{(0,1)}, s_0^{(0,1)}, s_5^{(0,1)}, s_5^{(0,1)} \rangle, \\ s_2^{(0,2)} &: \langle s_2^{(0,1)} \rangle \end{aligned}$$

It ends up being the same pattern of construction this way: You continue adding sequences to new cells to the right of the previous one—using names from the previous one—and all of this, as long as you stay within the same row.

We continue this process, but at some point we want to move on, we want to construct the next row, we want to translate to constructing the next unified powertuple. We start with the first cell of the next row. In the table, all rows following the first have their initial cell as empty. The reason for this is that

$$\mathbb{T}^0(\mathbb{V}^n) \quad := \quad \mathbb{V}^n \quad := \quad \bigcup_{k \geq 0} \mathbb{T}^k(\mathbb{V}^{n-1})$$

which is to say the first cell translates to being the accumulated names of all the previous cells. As they already exist in the previous rows—and as this is a practical implementation—we only need specify them once without duplication where we first encounter them.

As for the next cell to the right (which we intuitively consider to be the *actual* initial cell for this row), here you again create sequences of names from the previous, but instead of the previous cell which is empty, you now have access to all the names within all the cells of the previous row. It doesn't stop there though: As you construct all such following "initial" cells in rows below, not only do you have access to the names of the previous row, you also have access to all the names within all the previous rows, to which you can mix and match to create new sequences.

After that, as you construct cells rightward within the same row, it's very much like the first row: You can only use names from the previous cell.

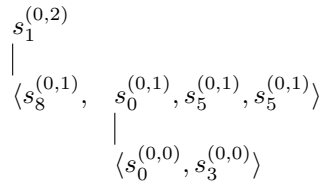
If you do things this way, then the whole structure of our stratituple corresponds (in potential) to the definition of the more theoretical stratified powertuple that inspired it.

Algebra

When we look to introduce an algebra for our stratituple, some subtleties of intepretation arise.

The main issue boils down to the fact that though our stratituple is a data structure in its own right, it can also be considered a *universal data structure*, which is to say it's meant to simulate all other finite computable data structures. In particular, any such simulated data structure internally would be expressed through the nested sequences we've described within. For example if you unravelled one of the previously mentioned example sequences, say $s_1^{(0,2)}$, you'd get [3] :

[3] Technically this example is incomplete, as names belonging to $\mathbb{V}^0\mathbb{T}^1$ only exist if they point to something within $\mathbb{V}^0\mathbb{T}^0$, though in this example we leave some of that information out for simplicity of concept.



Effectively, you have a bunch of tree structures, refactored to share resources to reduce memory usage (among other reasons). From a theory of sequences—where the game is to represent everything else in the finite mathematical universe as sequences—you can reinterpret these trees as any other type of data structure, such as a *set*, *list*, *graph*, *hash table*, etc. You might ask if that's even possible, but as it stands these constructs are already implemented in many well known programming languages for use within register machine architectures as address based memory (sequences), and although this is not a theoretical proof, it will here be taken for granted as close enough.

So when we speak of an algebra for our stratituple, what level of implementation are we looking to introduce its operators? Low level nested sequences? Optimized grammar for high level abstract semantic data types?

The level with which we've introduced the stratituple, is itself our solution. The main realization is that we introduced our stratituple by translating from the mathematical model. We furthermore explained how one would construct the cells (row by row, left to right, the first cell down has access to all previous names...), and it is at this level we introduce our operators of interest. At this level of implementation it should be noted we are again at a *midway*, a point of intersection between specification and implementation: Operators that focus on introducing sequences into cells are the basis for any specification we may refine later on, but also offer enough flexibility themselves to still be decomposed and optimized for any actual architectural implementation.

To be fair, this is all a long winded way of saying we are focusing on operators which act on the stratituple as data structure itself. We will standardize our approach by means of considering its lifecycle, for which we've already informally described as part of its definition.

When focusing on the lifecycle aspect of it, I would say the main trends are: *birth, growth, change, decline, death*, not to sound too poetic or anything. As for a computing lens, this might translate to: *construction, extension, mutation, restriction, destruction.* As mentioned above, we are looking for midway operators, which are sufficiently powerful and expressive on their own, yet can be used to compose more structured operators representing the lifecycle changes of our stratituple.

construction

Construction would more or less consist of creating a dynamic list (rows) of unified powertuples, where each row is itself a dynamic list (cells) of regular powertuples, where each cell is then a dynamic list of *sifter* names [4] . It's kind of like a 3-dimensional nested dynamic list. Such structures (N-dimensional) I prefer to call *plots*.

Otherwise each cell would start empty. Keep in mind construction is effectively initialization, so we create the overall structure and leave it in a state ready for extension.

destruction

Destruction would mean destroying each dynamic list accordingly, which is to say deleting all the sifter names within cells so that we return to an empty structure, then destroying the structure.

extension

Extension in the broadest sense is to supply a sequence of sifter names to the stratituple, and request that the sequence be added to the appropriate cell.

In this situation, our operator dispatches according to sequence *type*. Interestingly, the means to which we've created cells within the stratituple table supplies us with a natural type system for which we now

[4]The idea of a *sifter* will be elaborated upon in part three.

compare these structures. Two sequences have the same type if they belong to the same cell. As such, the idea of *cell matching* and any related subroutines are instrumental to the extension operators we specify.

Our operator then would start by taking its input sequence and analyzing it to know which cell it belongs to. In order for it to be successfully added, we would then verify that the sifter names that make up its components each belong to previous cells. In such a case, before we add it to the appropriate cell, we'd first check if it already belongs. If not, then we could finally add it.

restriction

Restriction is more subtle as it has an important constraint. Namely, in theory when looking to remove a given sequence, you should be able to specify its cell location and its name, and if it's there it would be removed. The subtlety comes from the fact that other sequences in other cell locations may depend on this sequence (rather its name), and so if removed, you'd have a dangling pointer more or less.

There's no single best policy to handle this: We have a point of modular divergence in the design of our stratituple specification. I myself have chosen a design such that sequence removal **is allowed**, but only if no other sequences depend on the sequence in question.

This actually suggests an extension to our stratituple to include count information embedded in each name. [5] This of course is an optimization, and could be considered later, but I thought it worth introducing here anyway.

mutation

With a design policy that allows for restriction, we also now have the possibility of mutation. Technically it exists because in theory you can remove a sequence, and then add another sequence in the same location with the same name. Strictly speaking these are separate actions, but in effect they could be interpreted as mutation. In which case, it makes sense to allow it so that it can be optimized accordingly, and in particular: *safely*. Of course its inclusion would be constrained under the policy that you can only mutate a sequence if no other sequence in any other cell depends on it.

Conclusion

Now that we have our weakly specified stratituple data structure, how do we implement it into a computational model of reference?

Turing machines are effective because universal machines exist—machines which simulate all other Turing machines. The greatest strength of our stratituple as a data structure when used toward a theory of computation is that it is a *universal* structure—a structure which simulates all other data structures. How is this relevant to computation? We now have a medium to navigate any data structure in a unified way. It gives us a universal grammar to reference content and express the ways in which we wish for that content to change computationally.

It may not be clear yet, but this opens up new ways to express all existing computational concepts used in the design of a modern programming language, which offers a foundation to a new genre of languages in general. This of course will be the topic of *part three*.

Pijariiqpunga.

[5] Such a counter system is reminiscent of the pushdown automata of a context free grammar. It might be worth researching whether such an implementation has a context free grammar embedded within.