Datra primer

`@qualiascript` ChatGPT

This post is meant to serve as context and intuition-building for the Datra preprint, and as such, it will take a different approach to Datra. We will be focusing on the StaDaTraMon monoidal category, but instead of introducing it in technical terms, I will illustrate the ideas underpinning the design of the Datra programming language. I will start from the original vision, wherein Datra was supposedly a language where "everything is a map". This idea has not survived the final design, however, the idea of maps is still central to Datra and serves as a useful starting point.

Let us define sets $A = \{1, 4, 7\}$, $B = \{2, 3, 5\}$, $C = \{6, 8, 9\}$. Then, `[[A; B]; C]` serves as a Datra map, which is distinct from `[A; [B; C]]`, `[B; C; A]` or other maps with the same underlying data but a different presentation. That is because, for instance, `[[A := 1; B := 5]; C := 8]` serves as an inhabitant of the original map, but not of the other maps, despite obviously being able to be turned into an inhabitant of the maps with the same data. We also consider map `[[A; B]; C; []]`, where `[]` holds no data, however, we deem this map as equivalent to `[[A; B]; C]` by definition. The map `[A; [B; C]]` can be viewed as a tree, in the following manner:

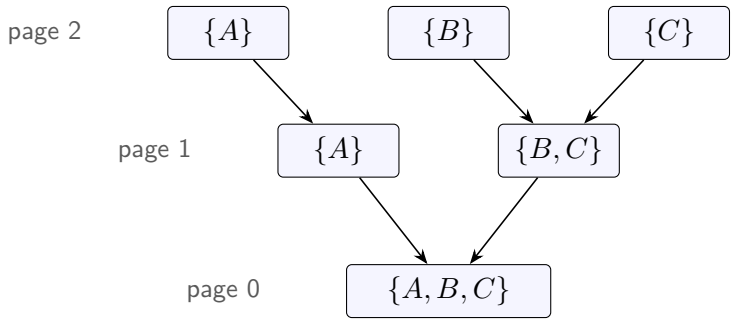

Figure 1: The pagination of `[A; [B; C]]`.

Things to note in this tree structure: its cardinality is the height of the three, in this case, 3, from 0 to 2. The collection of nodes at each height level is considered a page, and by extension, the shape of the three, without the data underlying it, is considered a pagination. At page 0, the tree has only one node, whose underlying data we consider the **extent** of the DaTra type. The nodes on the last page are considered the **territory**. The nodes on each page are ordered, and the edges between nodes respect this ordering, as we established previously, despite any ordering having isomorphic data. Going from a node on page x to page $x - 1$, the edge must represent an injection, as we retrieve some, but not necessarily all, of the data. Furthermore, the images of these injections ought to be pairwise disjoint in the target, as to not generate new data that was not available in the extent.

For the tree structure to represent a map, there is another requirement: the territory must match the extent, as in, all data in the extent mus recall that t be represented by an argument of the territory. Otherwise, the supposed map would have inaccessible "hidden" data, not represented by its literal notation, such as `[A; [B; C]]`. This is, however, completely fine, as long as we do not take for granted that the structure must represent a map. An alternative view would be to see the tree structure as representing the different presentations of some data

at different pages, allowing for data loss in the process. This is a useful generalization of the concept, and one that we will take. Furthermore, given any atlas, one can recover a map from it by using the Charting functor, as detailed in the preprint.

Another issue to consider is size. In this example, everything is finite, but this is not necessarily the case. However, we also seek to avoid data that is too large to handle in concrete, computational terms. The solution taken is as follows: all the data at any node is to be countable, and each node is to have a countable number of nodes in the page above that point to it. The cardinality, that is, number of pages, is finite, however, we will find ourselves in the situation of asking for the data at a page number higher than the cardinality. In that case, we consider that the final page repeats. The nodes at each page are still ordered, however, they are not ordered merely by positive integers, but by ordinals smaller than $\omega_0^{\omega_0}$. This convention allows the ordering of nodes even after a countably infinite number of nodes are introduced.

To be specific, the data structure we have introduced so far is an atlas object, by the terminology of the preprint. An object in StaDaTraMon can represent multiple atlas objects, by usual sum types logic. Atlas morphisms are flexible, but morphisms in StaDaTraMon are more restrictive. Let us look at the case of two atlas objects represented by StaDaTraMon and a morphism between the two. The rules are as follows: each node from the origin atlas must be sent to a node of the target atlas, so that the function between atlas nodes is injective. The extent of the origin atlas must be sent to the target of the origin atlas. The ordering of nodes must be preserved, in the following sense: given any two nodes of the origin atlas, we can find whether one is to the left of the other by pulling back at a common page. Then, the morphism must respect this order in the image, as well. Each origin node sent to a target node must induce an injection between the data at the two nodes, that is, each node is sent to a node with more or equal data. Finally, for any node in the origin, if it has a territory in the final page that it is the image of, then this must be preserved by the morphism. Each of these rules can be relaxed by moving to a different category than StaDaTraMon, however, all these rules are imposed in StaDaTraMon itself.

To illustrate why these rules are helpful, consider a morphism from `*` to `[A; [B; C]]`, where `*` is the atlas given by a tree with cardinality 1 and whose extent is the singleton set. In this case, a morphism picks an element of A, B or C, thus viewing the map as a sum type. Also consider, for any atlas map, a morphism from the map with the same pagination, but whose territories are all singleton. In this case, the morphism selects an element of each of the values of the map, while remembering the full data. For instance, `[A := 7; [B := 2; C := 6]]` can be denoted as one of those morphisms.

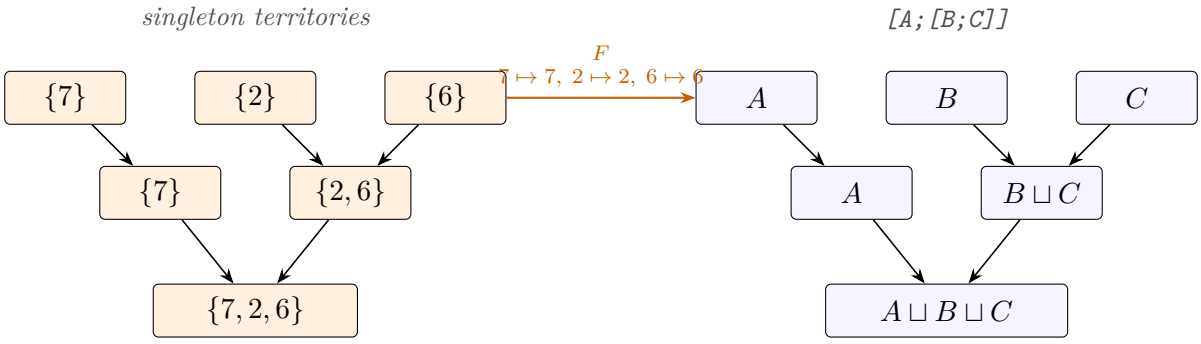

Figure 2: The atlas morphism denoted by `[A := 7; [B := 2; C := 6]]`. Orange nodes carry the selected singleton data; the horizontal arrow summarizes its nodewise injections.

As such, Datra semantics support partial and total maps, as well as canonical map orderings. By using multiple atlases and imposing a coherence condition, it can also represent maps who's some of the arguments admit multiple orders. Part of the underlying data can also include

argument names, represented by identifier strings that depend on values. This ties with another concept for Datra: a language for partially named partial maps with partially variable argument order. However, even this is a mere facet of Datra semantics, which are flexible enough to model various aspects of data modeling, function call semantics, config files and other programmatic aspects which fall into the broad category of "data transformations". In particular, by dropping the map requirements, an atlas can also model the results of computation at different stages, so that this unifies maps and functions within the same semantics.

The so-called "horizontal sum" operator, $\boxplus$, takes two StaDaTraMon morphisms and puts them side by side. It can be seen as the operation of merging two trees by letting the extent be the coproduct of their respective extents, and each page above 0 being composed of the pages of the trees at the respective level. This operation is closely related to the map-creating operation, for instance, one can construct `[B := 2; C := 6]` out of $(B := 2) \boxplus (C := 6)$. However, the map semantics instead keep the extents of the trees on page 1 of the new tree. Below is a representation of the horizontal sum operation on $(B := 2) \boxplus (C := 6)$.

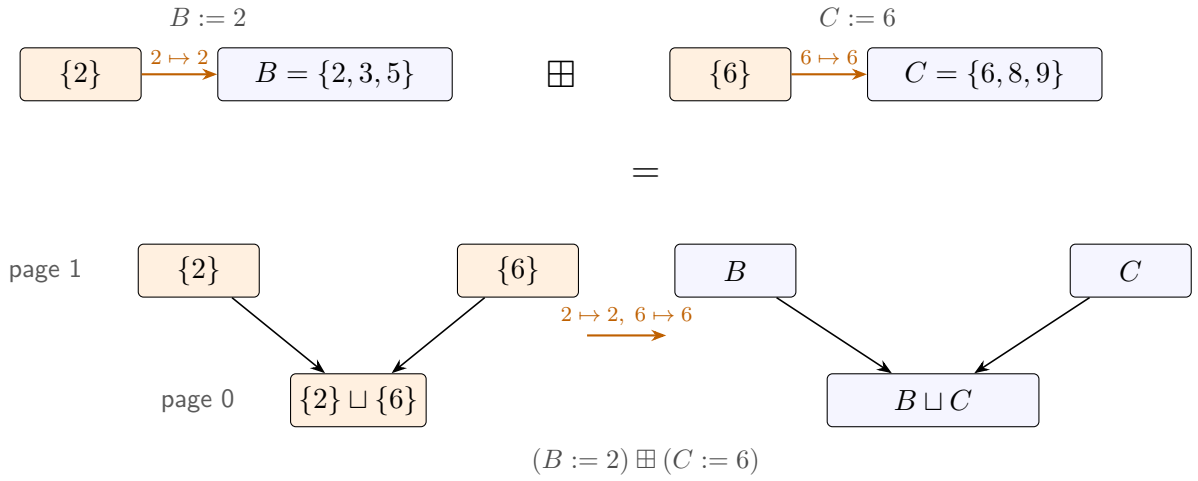

Figure 3: Each operand is a morphism between one-node atlases. Their horizontal sum forms a single coproduct extent on page 0, while page 1 contains the two operand pages side by side. The orange arrow summarizes the nodewise injections of the summed morphism.

I believe that there are many other interesting aspects of Datra semantics to explore, which become easier to comprehend as part of an actual programming language, rather than as manually-explained facets of the mathematical model of the language's semantics. This document serves as an explanation for the design principles underlying Datra, not a comprehensive explanation of the language's semantics and capabilities. For instance, there ought to be a way to model Python-style `kargs` / `kwargs` as first-class Datra constructions, however, this is beyond the scope of this document. The eventual construction of advanced Datra features is best kept within the actual Datra programming language, which I believe StaDaTraMon has enough flexibility to model in a type-safe manner.