Understanding the Python Prototypes: Multinomial Structures and Reasoning DNA Units

Eric Robert Lawson

October 24, 2025

Abstract

Although the Prototypes are based on rigorous mathematical work, it may still be confusing to not only understand the mathematics as someone who may not be a mathematician, but also to understand why the prototypes do not focus much on path transversal. In this document I will be going into detail how this is an aspect of the universality of the reasoning substrate, how your contextual intention for operationalization and manipulation of the RDU influences the POT (pruning-ordering-type) generator function, and how non-trivial, non-linear conclusions can arise as a result of systemic emergence.

Contents

1 Introduction

In this document, we provide an accessible overview of the Python prototypes for symbolic reasoning and DAG-based computation. We discuss how the prototypes implement layer-wise operations, the principles behind Reasoning DNA Units (RDUs), and how intended vs. unintended operationalizations can lead to non-linear emergent behavior. This overview is designed for readers without a deep background in formal mathematics.

2 Non-mathematical understanding of multinomial structures

To understand the deeper structure shared by all prototypes, we must first understand what it means for reasoning itself to have a combinatorial or multinomial form.

My prototypes explore several ideas, but the one thing that unites them is the concept of a *multinomial structure*. But what does that mean?

Let's think about structure using real-world examples and concrete possibilities.

Suppose we have 3 apples and 2 bins. We want to know all the ways the apples can be arranged into the bins. Each arrangement represents a distinct possibility of how many apples go in each bin.

- Bin 1 has 3 apples, Bin 2 has 0 apples:

$$(3, 0)$$

- Bin 1 has 0 apples, Bin 2 has 3 apples:

$$(0, 3)$$

- Bin 1 has 2 apples, Bin 2 has 1 apple:

$$(2, 1)$$

- Bin 1 has 1 apple, Bin 2 has 2 apples:

$$(1, 2)$$

These four possibilities represent all distinct ways to distribute 3 apples across 2 bins.

This is an example of a multinomial structure of order 3 in 2 dimensions. In mathematics, it's often called *binomial* because there are two bins, but the idea generalizes: *multinomial* simply means there can be more than two bins.

More generally, if we have n items (the *order*) and k bins (the *dimensions*), the set of all possible distributions defines a *multinomial structure*.

3 How do these concepts relate to the prototypes?

Given the structure just described, we can now see how this directly applies to the Python prototypes.

3.1 Multinomial expansion

In mathematics, a multinomial expansion of order n and dimension k is represented as:

$$(f_1 + f_2 + \cdots + f_k)^n$$

For example, in the case of order $n = 3$ and dimension $k = 2$, we have:

$$\begin{aligned}(f_1 + f_2)^3 &= (f_1 + f_2)(f_1 + f_2)(f_1 + f_2) \\ &= \binom{3}{0} f_1^3 f_2^0 + \binom{3}{1} f_1^2 f_2^1 + \binom{3}{2} f_1^1 f_2^2 + \binom{3}{3} f_1^0 f_2^3\end{aligned}$$

The structural patterns here correspond to the same tuples as before:

$$(3, 0),\ (2, 1),\ (1, 2),\ (0, 3)$$

3.2 Higher-order derivatives of product of multiple functions

In calculus, we often take derivatives of products of functions. The nth-order derivative of a product of k functions is written as:

$$\frac{d^n}{dx^n}\big[f_1(x) \cdot f_2(x) \cdots f_k(x)\big]$$

For example, for $n = 3$ and $k = 2$:

$$\frac{d^3}{dx^3}[f_1(x) \cdot f_2(x)] = \binom{3}{0} f_1^{(3)}(x) f_2(x) + \binom{3}{1} f_1''(x) f_2'(x) + \binom{3}{2} f_1'(x) f_2''(x) + \binom{3}{3} f_1(x) f_2^{(3)}(x)$$

Again, these derivative combinations correspond to the same structural tuples:

$$(3,0),\ (2,1),\ (1,2),\ (0,3)$$

This is exactly what is implemented and visualized in the Python prototypes.

The multinomial pattern we just explored also underlies many structures in advanced mathematics. One particularly important family are the Bell polynomials, which encode how composite derivative structures distribute across partitions of function orders. These are central to the symbolic mechanisms implemented in the Python prototypes.

3.3 Convoluted and Typical Partial Bell Polynomials of the Second Kind

In Kurchinin's work on the derivation of Bell polynomials of the second kind, these functions can be expressed as coefficients extracted from a multinomial generating function:

$$B_{n,k}^f(x) = \frac{n!}{k!}[z^n]\big(f(x+z) - f(x)\big)^k$$

From a non-mathematical perspective, the main conceptual difference from the earlier multinomial case is that here we *exclude* situations where a bin has no apples at all. Additionally, the *order of the bins does not matter*. This means that if Bin 1 has 1 apple and Bin 2 has 2 apples, or vice versa, we treat both as the same configuration — a single unique structural object.

Taking this into account, we are left with only one structural pattern: $(2,1)$ (or equivalently $(1,2)$ since order does not matter). For example:

$$B_{3,2}^f(x) = 3f'(x) \cdot f''(x)$$
$$(2,1)$$

Now, what does this mean in terms of the **Convoluted Partial Bell Polynomial**? If we consider every possible moment in which order 3 can be distributed across two dimensions, we obtain:

$$3f_1'(x)f_1''(x) + 3f_2'(x)f_1''(x) + 3f_1'(x)f_2''(x) + 3f_2'(x)f_2''(x)$$
$$(3,0),\ (2,1),\ (1,2),\ (0,3)$$

Here, we can clearly see that a multinomial pattern is *embedded within* another multinomial pattern — this is what gives rise to the convoluted partial Bell polynomial structure.

3.4 Structural Relational Conclusion and How It Connects to Prototype Examples

There is a shared structural foundation underlying all four prototypes. However, in conventional mathematics, these ideas are rarely treated as directly connected in origin. Through reasoning-based deduction and manipulation of reasoning proof objects, we can see that these links arise naturally.

In the context of the POT generator function within the Python prototypes, the probabilistic outcomes — represented by primitives such as $(3, 0), (2, 1), (1, 2), (0, 3)$ — may appear to simply reflect combinations and permutations. Yet, this is not merely combinatorics; it is a **structural relational mapping** between the multinomial framework and the POT generator function itself, in direct correspondence with the creation of RDUs and the layer collection operation.

This structural linkage is what makes the prototypes both mathematically grounded and universally extensible: they reveal how reasoning operations can be expressed, composed, and emergently manipulated through multinomial and Bell polynomial architectures.

4 Intended Operationalization of RDU Influences POT Generator Function

As mentioned briefly in the previous section, a defining property of Reasoning DNA Units (RDUs) is that the way an RDU is operationalized directly influences the POT generator function. In turn, the POT generator function also shapes the operational purpose and the resulting transformations that can be performed on the RDU. The **POT generator function** (Pruning–Ordering–Type) defines the structural rules that govern how reasoning operations unfold within an RDU. It determines which nodes are preserved (pruning), in what sequence operations occur (ordering), and what kind of transformation applies (type). It is important to note that the interpretation of pruning, ordering, and type can vary depending on the context. For instance, in the context of path transversal, the "ordering" may represent the priority of which decision to explore along a path, with the first in line reflecting the highest-priority choice (akin to an initial intuition).

In simple terms: how we *build* the RDU affects what operations can later be *done* with it, and vice versa.

For instance, the Python prototypes primarily utilize the layer collection operation when manipulating the Reasoning DNA Unit. (Other than path transversal being useful for the Convoluted Partial Bell Polynomials to consider slices of convolution moments) This is because the creation of these RDUs was done considering the fact that I would have to collect each node on the layer rather than performing a path transversal operation to obtain the final result.

That is not because I cannot construct a RDU for which I can construct these results from path transversal, but because this allows for utilization of path transversal to have a different meaning when observing reasoning with the consideration of this operationalized context!

Why is this important?

This shows that reasoning itself can be formalized as a *constructive process*—not merely a static proof but a system of operational relationships.

This is the systematic formalization of discovering non-linear structural relations in reasoning! These relations arise when operations originally defined for one reasoning context reveal unexpected symmetries or transformations when applied to another. Here reasoning and the operationalization of reasoning during its construction can enable you to perform other operations on the proof objects

to obtain different results. Also considering the python prototypes include composition of RDUs, these relations can nest within each other and elevate these non-linear emergent relations!

Convoluted Partial Bell Polynomial Example Is Also Example Of Compositional Use of RDUs

When observing the Python prototype code for Convoluted Partial Bell Polynomials, we notice a key property: one RDU is composed *within another* through sub-DAG placement during layer collection. That is, within a single layer collection operation, the output of one RDU serves as the input for another.

This shows that RDUs can be composed hierarchically and that reasoning itself is inherently compositional in the same way. However, this influences the meaning behind layering, also making the composed (or nested reasoning process) a holistically defined sub-process in the reasoning object.

Take, for instance, in the prototype for Convoluted Partial Bell Polynomial, it was the convolution (or mixing up) of a number of entities (in this case arbitrary functions) **within a fixed structure of partial Bell Polynomial nested structure**.

Example: Non-Linear Relation via Path Transversal and Layer Collection

Consider the example of constructing a chess RDU, as discussed in another article (*Toward a Domain-Specific Language for Reasoning: A Practical Introduction to Reasoning DNA Units*).

We can define a Reasoning Proof Object for chess such that:

- **Combinatorial layering:** the sequence of chess gameplay, move by move.
- **POT generator function:** the set of all legal moves available at a given game state, conditioned on the current path transversal.
- **Path transversal:** the act of playing through the chess game layer by layer, move by move.

What would layer collection at a given path transversal point look like? Well in the context of a chess game, if you perform layer collection at a specific point in path transversal (during a game at specific move set), that would enable you to perform an operation on every possible potential move.

This would allow you to potentially measure how good a potential move is in any given game state. Just like a modern chess engine shows what it thinks is the next best possible move in ranked order given a position on the board.

This is where the utility and non-trivial results can really shine through, when performing unconsidered operations on reasoning objects, operations not fully anticipated during POT generator contextualization. Here we can formalize non-linear systemic relations!

Conclusion

In summary, the RDU framework provides a way to treat reasoning not as a linear path but as a dynamic system of compositional operations. By doing so, we can discover non-linear structural relations between reasoning processes—relations that emerge only when reasoning itself becomes an object of reasoning.

Version Note

This document is part of the *Reasoning DNA Unit (RDU)* documentation set accompanying the Python prototypes for symbolic reasoning, authored by Eric Robert Lawson.

For a full conceptual context, it is recommended to first read:

- **The Manifesto** – philosophical and historical foundations of the framework.
- **The DSL Roadmap** – outlines the design trajectory for a domain-specific language for reasoning.
- **A Practical Introduction to Reasoning DNA Units** - provides the foundational definitions, structures, and examples.