From Haskell Traces to Causal States: An Empirical Derivation

In this document, we look at how an infinite dataset of Haskell execution traces naturally reveals the "causal state" structure that computational mechanics predicts. We show how these causal states correspond to *Haskell types* and how a language model (such as a transformer) trained on these traces effectively learns the minimal state required to predict further outputs—making it an *optimal* predictor under certain conditions.

Before diving in, it's worth recalling some key points from the broader theory:

1. Causal States: In computational mechanics, the minimal "lumps" of history that preserve full predictive power are called *causal states*. Two histories that predict the same future distribution belong to the same causal state.

2. Typed Functional Programming: In strongly typed languages like Haskell, the type system imposes structure on which function can follow which output. This often matches the notion of "causal state" because being in a type τ tells you exactly which functions (and thus transitions) are possible next.

3. Implicit vs. Explicit Composition: Language models can either - *explicitly* generate code that composes functions, or - *implicitly* predict what a function call *would* produce without explicitly writing out the function call.

   Despite their different styles, both approaches rely on learning the same underlying states (type-based or otherwise).

With those big ideas in mind, let's now go section by section through the empirical derivation.

1. The Infinite Haskell Trace Dataset

Context: We start with the idea of an *infinite dataset* D containing execution traces from *all possible* Haskell programs. Each trace is a sequence of steps:

$$T = [(f_1, x_1, \tau_1, y_1), (f_2, x_2, \tau_2, y_2), \ldots, (f_n, x_n, \tau_n, y_n)]$$

where each step has:

- f_i: the function being applied,
- x_i: the input,
- τ_i: the type signature in play at that step,
- y_i: the output of that function call.

Why This Matters:

1. Such a dataset captures not just static code, but the *actual runtime behavior* of Haskell programs.
2. Because Haskell is strongly typed, each step (f_i, x_i) must align with the type τ_i.
3. Over an *infinite* dataset, we see *every possible valid composition* of functions (plus invalid or error-prone ones, if we record them).

Example A typical snippet might show JSON parsing and subsequent calls:

```
[
  (parseJSON, "{\"user\":\"john\",\"age\":30}", String -> Maybe User, Just (User "
  (getAge, Just (User "john" 30), Maybe User -> Maybe Int, Just 30),
  (isAdult, Just 30, Maybe Int -> Bool, True)
]
```

This snippet shows: - The function ‘parseJSON‘ was called with a ‘String‘, producing a ‘Maybe User‘. - Then ‘getAge‘ was called on ‘Maybe User‘, producing a ‘Maybe Int‘. - Finally ‘isAdult‘ was called on ‘Maybe Int‘, producing a ‘Bool‘.

Hence each step includes *function, input, output, and type signature* in the trace.

2. Properties of the Trace Distribution

Before jumping into the formal properties, let's remember *why* we care:

- We know Haskell's type system *strongly restricts* which function can come after which output. This makes the chain of steps *type-driven*. - The distribution of valid traces is shaped by *value-specific paths* (the actual data influences which path is taken) and *common compositional patterns* (like using ‘map‘ followed by ‘filter‘, etc.).

2.1 Type-Driven Transitions

Key Idea: If you end a step with a value of type τ, that strongly determines which next functions are *even possible*. In formula form:

$$P(f_{k+1} \mid y_1, y_2, \ldots, y_k) \;=\; P(f_{k+1} \mid \text{type}(y_k)).$$

Interpretation: Because Haskell is statically (and strongly) typed, if your last output was a ‘String‘, you cannot pass it next to a function expecting ‘[Int]‘ unless you do something that changes type in between. So the distribution of next functions depends almost entirely on the type of the current output.

2.2 Value-Specific Paths

Key Idea: Although the *type* heavily constrains possible next functions, the *value* also matters. For instance, if a function might produce different results depending on the input data, the path branches in that sense:

$$P(y_{k+1} \mid f_{k+1}, x_{k+1}, y_1, \ldots, y_k) \;=\; P(y_{k+1} \mid f_{k+1}, x_{k+1}).$$

Interpretation: Haskell's referential transparency and deterministic evaluation mean that once you *know* which function f and which input x are used, the result y is determined (barring randomness or side effects). The distribution simplifies to “what does $f(x)$ yield?”

2.3 Compositional Patterns

Key Idea: Over an infinite dataset, some patterns occur over and over—like “‘map‘ a function, then ‘filter‘,” or “parse some JSON, then check fields.” These patterns reflect typical usage in real Haskell code.

Interpretation: This repetition ensures the model sees repeated examples of the same "macro-structure." If you see '(map, [1,2,3])' followed by '(filter, ...)', the next step often might be '(fold, ...)', etc.

2.4 Error Distributions

Key Idea: Even though Haskell is statically typed, we can still see *runtime errors* (e.g., partial functions like 'head []') or type boundary mistakes in ill-typed examples (if the dataset includes them, say from logs or attempts). Typically these appear where the *type* or *value* is incompatible (like calling 'head' on an empty list).

Interpretation: From a distribution viewpoint, these errors become part of the "tails" or rare events that might or might not be included, depending on the dataset's coverage.

3. Inducing Causal States Empirically

Rationale: Causal states (from computational mechanics) are all about *partitioning* the space of histories in such a way that everything in one partition (one "state") leads to the same future distribution. We want to see if—just by looking at these traces—this partition emerges and matches Haskell types.

3.1 The Empirical Equivalence Relation

Definition: Two trace prefixes P_1 and P_2 are equivalent if they yield *identical* conditional distributions over all possible continuations:

$$P_1 \sim P_2 \quad \Longleftrightarrow \quad P(\text{continuation} \mid P_1) \;=\; P(\text{continuation} \mid P_2).$$

Interpretation: This is the standard "predictive equivalence" that defines a causal state: if two prefix histories lead to the exact same distribution of future outcomes, they're in the *same state*.

3.2 Type-Based Partitioning

Observation: Empirically, if you group all prefixes that end in a value of type τ, these appear to have the same distribution of possible next functions, next values, etc.

$P(\text{continuation} \mid \text{prefix ends with type}(\tau_1)) = P(\text{continuation} \mid \text{prefix ends with type}(\tau_2)) \quad \text{iff} \quad \tau$

In other words, "if you end with 'String', you face the same set of next-step distributions as all other prefix histories that ended in 'String'."

Interpretation: This matches the strong type constraints in Haskell: *knowing the last output's type suffices to predict the distribution of what's next.*

3.3 Equivalence to Causal States

Key Result: These type-based equivalence classes fulfill the computational-mechanics definition of causal states:

1. Sufficiency: Knowing only "I'm in a type τ" is enough to predict the next-step distribution.
2. Minimality: If you lump different types together, you lose predictive power. For example, conflating 'Maybe Int' with 'Int' changes the possible next function set.

Hence, from the dataset alone, you discover "types as states," which is the same conclusion the *theoretical* approach would yield.

4. Probabilistic Transition Structure

Why This Matters: Having identified that "states = types," the next step is to see how states transition to one another. This is basically the "Markov chain" or *-machine* transition function—except that in Haskell, it's strongly determined by the function and input.

4.1 Transition Probabilities

Definition: For two types τ_1 and τ_2, the probability of moving from τ_1 to τ_2 is:

$$P(\tau_2 \mid \tau_1) = \sum_{f,x,y} P(f, x, y \mid \tau_1)\, \mathbf{I}[\text{type}(y) = \tau_2],$$

where $\mathbf{I}$ is an indicator function. Essentially, you sum over all function calls (f, x) that produce an output y whose type is τ_2, given that your current type is τ_1.

Interpretation: This captures how, *empirically*, we move from one type to another by applying some function f to some input x.

4.2 The Unifilar Property

Unifilar: A *unifilar* machine is one where, if you know the current state and the next "symbol" (or "input," or "tool"), the *next* state is *deterministically* determined. In Haskell's deterministic evaluation:

$$P(\text{next state} \mid \text{current state}, \text{function}, \text{input}) \in \{0, 1\}.$$

So effectively, "Given I'm at type 'Int', and I apply function 'f :: Int -> Bool' with a certain input, I'll definitely end in type 'Bool' next."

Interpretation: Unifilarity is crucial in computational mechanics for easy state tracking. It also ties into how Haskell's type system is *deterministic*, meaning once you specify your function and input, there's exactly one resulting type.

5. The Minimal Predictive Model

Motivation: Now that we see how states (types) transition, we can build a minimal model—an *-machine* or *transducer*—that captures exactly the structure discovered in the dataset.

5.1 State Space

Definition: The states S are simply *the set of all types* τ we see in the dataset. Because we said the dataset is infinite and covers "all of Haskell," we have "all possible types" (in practice, an unbounded set, though any given snippet is finite).

5.2 Transition Function

$$T(\tau, f, x) = \text{type}(f(x)),$$

assuming x is of type τ. This just says "If you're in state τ, and you apply function f with argument x, you move to type($f(x)$)."

5.3 Output Distribution

$$P(y \mid \tau, f, x) = P(y \mid f, x) \quad \text{for } x \in \tau.$$

Meaning, once you specify the function and the input, the next output y is pinned down (or at least distributed in a known way).

5.4 Equivalence to -Machines

Interpretation: We basically have an *-machine* that's a function of "current type τ" plus "function + input." Because the transitions are deterministic, it is a unifilar machine. In short, *purely from data*, we reconstructed the computational-mechanics idea that "states = types."

6. Transformer Models Learn the Empirical Causal States

Connection to LLMs: When you train a transformer on these Haskell traces—token by token or step by step—it must learn a representation that *predicts* the next token or the next output.

6.1 Predictive Objective Forces Causal State Learning

Why: The transformer tries to minimize its prediction error. If the best partition of the past into "equivalence classes" for the future is precisely the type-based partition, the network *must* learn to track which "type" it's in.

Implicit vs. Explicit:

- Even if the model only *implicitly* outputs the function's result, it's forced to figure out: "Given I'm in the state representing 'Maybe User', which function can be next, and what's that output's distribution?"
- If it's doing *explicit composition*, it must likewise generate code that type-checks, leading to the same underlying state representation.

6.2 Attention Mechanisms Implement Transition Functions

Interpretation: Because each transformer layer can "focus" on different parts of the context, it effectively zeroes in on "the last output, and its type." This is how it infers "my next possible function calls are. . . ." So the attention mechanism is how unifilar transitions get tracked.

6.3 Hidden Representations Encode Types

Empirical: If you do clustering on the hidden states from a trained model while it processes Haskell traces, you'll find that *similar hidden vectors* correspond to "ending with the same type." That's a sign it has internalized the type-based state partition.

7. Empirical Results Support Theoretical Predictions

Two-Layered Claim:

1. We discovered from data that states = types.
2. Theory says that in a strongly typed language, "types" are indeed minimal causal states.

7.1 Empirically Derived vs. Theoretically Predicted Causal States

They line up perfectly: what we see from the infinite dataset matches the standard computational-mechanics argument that "if two histories yield the same distribution of futures, they're in the same state." In Haskell's case, "that state is the type of the current expression."

7.2 Minimality Property

If you try to collapse different types into the same "state," you lose predictive power (e.g., "'Bool' vs. '[Bool]' obviously allow different next-step

behaviors"). Hence you can't make a coarser grouping without hurting predictive accuracy.

7.3 Sufficiency Property

Similarly, no *finer* partition is needed than "type," because Haskell's strong typing ensures that everything else about the history is irrelevant for which next function can apply (barring the actual *values*, but even that is purely determined once you fix the function). So "type" is a *sufficient statistic* for describing the next-step distribution.

8. Practical Implications

Why This Matters: A purely theoretical result is nice, but we also want to see how it helps us train better models or design better systems.

8.1 Optimal Training Data

If you want an LLM to truly learn these causal states, you want data that *shows function calls and their results*. Execution traces—with explicit "(function, input) -> output" steps—are the best way. This is reminiscent of how "tool output prediction" (implicit composition) fosters the *optimal* internal representation for selecting the next tool.

8.2 Type-Awareness in Models

Because the model sees typed transitions, it effectively becomes *type-aware* even if it never explicitly sees "`:: Int -> Bool`" in text. The data's structure alone is enough to embed type constraints in its hidden states.

8.3 Predictive Power vs. Memory Usage

Computational mechanics says that these causal states use minimal memory *while* preserving predictive power. For an LLM, that means focusing on "the type of the last output" is near-optimal for deciding the next token or function—any coarser or finer partition is suboptimal in some sense (predictively).

9. Conclusion: From Data to Theory

What We Did:

1. We took an *infinite dataset* of Haskell traces, capturing how actual code executes.

2. By analyzing its statistical properties (which transitions are valid, how types shape next-step possibilities), we naturally derived the notion of *causal states*.

3. Those states turned out to be exactly "the type of the current value"—the same conclusion reached by the formal theory.

4. This explains why language models trained on code can learn to do correct function composition: they are effectively discovering these minimal predictive partitions.

Further Connection:

- This mirrors the broader theoretical results on LLMs as ε-transducers and how implicit output prediction ensures an agent learns the right "state" structure for *optimal tool selection*.

- In a typed environment, that means the model's internal states line up with the type system, letting it chain functions in correct ways—whether it *explicitly* writes the code or *implicitly* predicts the result.

Implication:

- Because we see the theory "emerge from the data," we confirm that the computational-mechanics perspective isn't just an abstract principle but genuinely matches how typed functional programs behave—and how an LLM can effectively master them.

Thus, both from a theoretical and empirical angle, we see that "types = causal states" in Haskell, and that LLMs will discover and exploit this structure when trained on execution traces, ultimately explaining why they can *so effectively* generate and reason about functional code.

Final Word

In sum, by simply observing the infinite set of valid Haskell computation traces, we arrive at the same partition—"states are types"—that abstract theory predicts. That synergy between data-driven discovery and theoretical frameworks is precisely what enables modern LLMs to act as powerful, *type-savvy* code generators and tool users.