Phantom Types and Interpreters: The Big Ideas in PSD3

The Big Ideas

1. **Phantom Type State Machine** - The `Selection` type carries phantom types that encode what operations are valid at each stage (data bound vs unbound, entering vs updating, etc.). This creates compile-time guarantees about D3's runtime state.

2. **Tagless Final / Free Monad Interpreters** - The same DSL can be interpreted multiple ways: to actual D3 JavaScript, to documentation/English, to Mermaid diagrams, to a MetaAST. This separates *description* from *execution*.

3. **D3 Join (Enter/Update/Exit)** - The core D3 pattern for data binding, but elevated to a first-class concept with type safety.

4. **Attribute System Evolution** - The progression from basic attributes to `v3Attr` with the TreeBuilder shows a pattern of making the DSL more declarative/compositional.

Two Kinds of "Tagless"

There's something funny about the fact that both the interpreters and the phantom types are "tagless" in different ways:

Tagless Final (the interpreters)

- "Tagless" means no runtime type tags / sum type constructors
- Instead of `data Expr = Lit Int | Add Expr Expr` with pattern matching
- You have `class Expr f where lit :: Int -> f; add :: f -> f -> f`
- The "tag" is the *choice of interpreter* at the call site, not a runtime value

Phantom Types (the state machine)

- Also "tagless" in that the type parameter carries no runtime data
- `Selection d s p c` - the `s`, `p`, `c` are ghosts
- They exist only to constrain what operations typecheck
- At runtime: `unsafeCoerce` because the tag was never there

Both are about **moving information from runtime to compile time**:

	What's erased	What remains
Tagless Final	Constructor tags	Polymorphic dispatch
Phantom Types	Type parameters	Valid operation sequences

And there's a pleasing symmetry:

- Tagless Final lets you write code *once* and interpret *many ways*

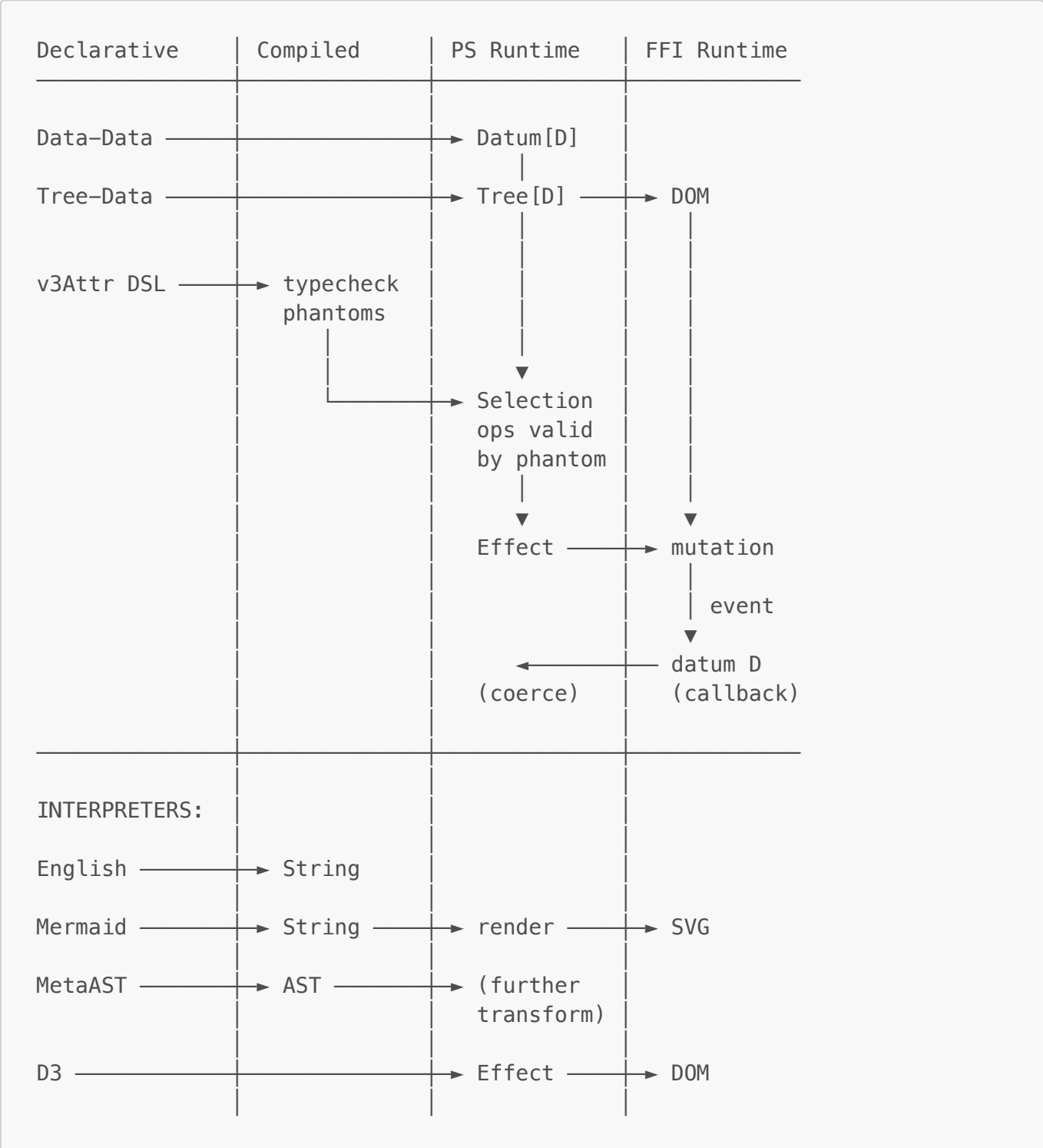
- Phantom Types let you write *many operations* that compose into *one valid path*

Both are ways of saying "the structure is in the types, not the values" - but from opposite directions. Tagless Final abstracts over *what you do with* the structure. Phantom types constrain *what structures are legal*.

The library uses both as a pincer movement on the same problem (type-safe D3) from two angles.

The Swimlane Architecture

The phantom types are about the relationship of the two runtimes (PureScript and FFI). The interpreters are about the relationship of declarative forms to static and dynamic outputs.



The phantom types live entirely in that "Compiled" lane - they're the membrane between Declarative and PS Runtime, ensuring only valid paths cross.

The interpreters fan out across lanes differently:

- English/Mermaid stay mostly static (Declarative → String)
- D3 interpreter crosses all the way to FFI Runtime
- MetaAST stops at PS Runtime for further manipulation

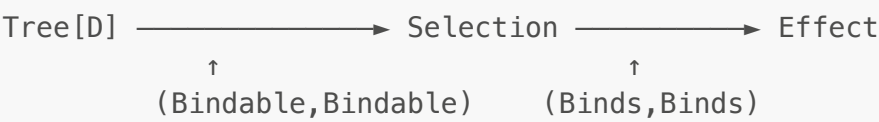
The `coerce` on the callback is that dotted line crossing back from FFI → PS Runtime, "morally correct" because the whole column shares D.

State Machine Duality

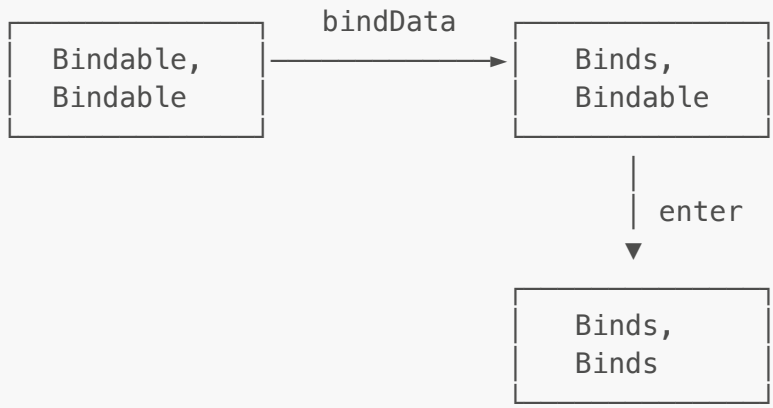
You can view the phantom type state machine in two equivalent ways:

Arrows as states (the "flow" view)

- Nodes are data/transformations
- Arrows show flow
- Phantom types annotate arrows: "this transition is valid because `Selection _ Binds _ _`"



Arrows as operations, nodes as states (the classic state machine view)



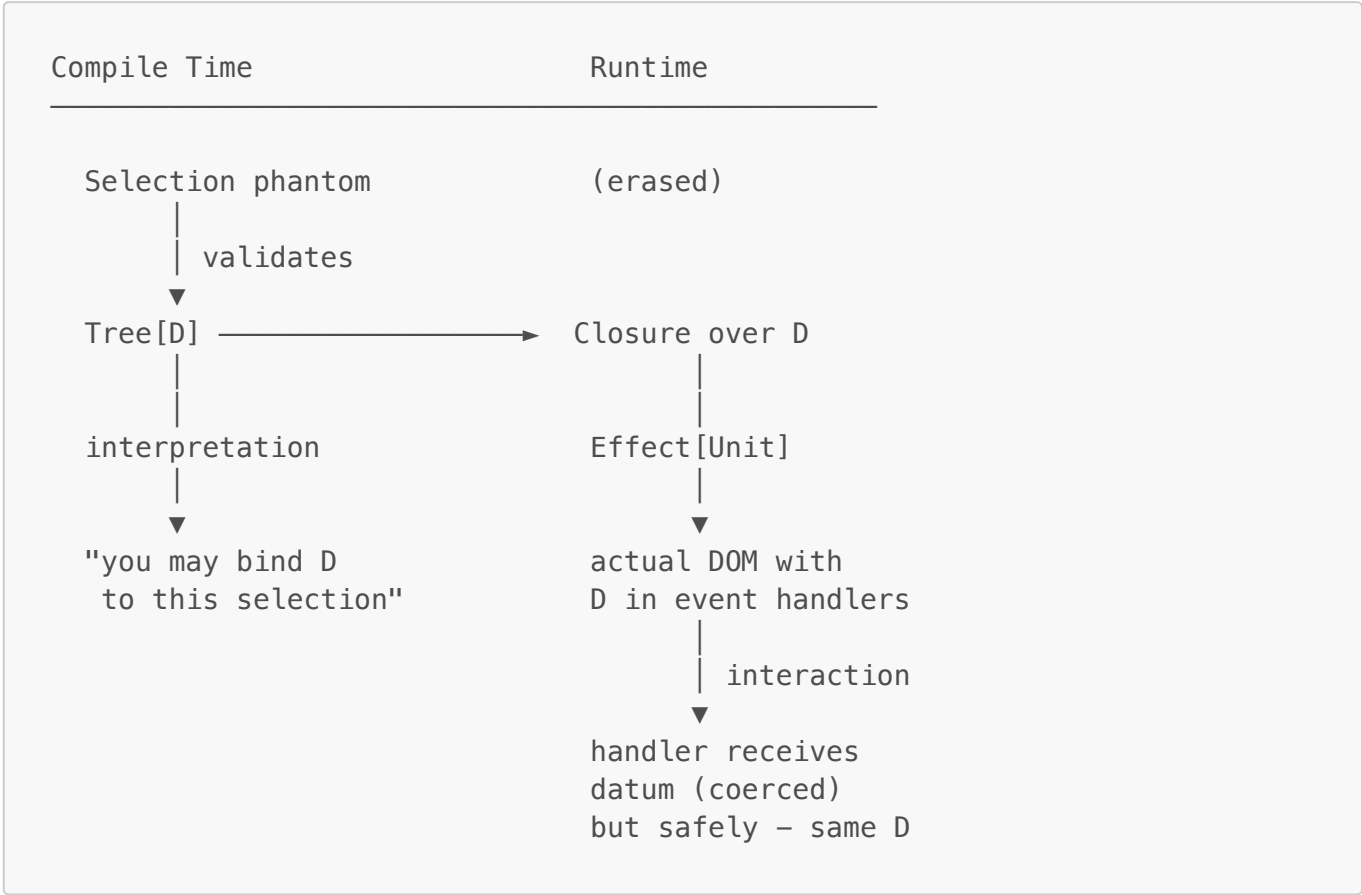
The first form is what you *actually experience* writing code - data flows through, accumulating phantom constraints. The second is the formal state machine.

The Role of Phantom Types (Subtle but Crucial)

The Machine (PSD3) tags the data it feeds into the Effect, which enables handling interactions because that tag comes back. It doesn't *really* come back - we use coerce - but we do so morally correctly because the

whole interaction is in the closure of one interpretation of a Tree, parameterized by one data type.

The "moral correctness" comes from the closure: when you interpret a tree with data type `D`, the entire interaction lives within that scope. The phantom type is like a wristband at a concert - it doesn't *do* anything at the show, but it proved you were allowed in, and everyone inside shares that context.



Visualization Ideas

The Sankey Watermark as Logo

The Sankey watermark in TreeBuilder is almost logo-like because it captures *flow* visually:

- Multiple sources converging
- Transformation through a middle layer
- Multiple outputs diverging
- **Quantities preserved** (what goes in comes out, just reshaped)

That last point connects to phantom types - they ensure nothing is "lost" in the transformation. You can't accidentally skip a state or produce an invalid output.

Potential Showcase Visualization

A visualization of the phantom type state machine as an animated state diagram:

- Shows that visualizations are declarative and start with no type associated
- Demonstrates how we ensure `unsafeCoerce` is moral by tracking "untyped-ness" then "typed-ness" via phantoms
- Interactive: a button to advance through states

- Like a visual proof that you can't get a mismatched type despite type erasure at the FFI boundary

This would be simpler than trying to show everything at once - just focusing on the type safety story through the state machine.