PSD3 Principled AST Design

Status: Proposed **Date**: 2026-01-26 **Branch**: `feature/recursive-join`

Executive Summary

This document proposes a fundamental redesign of the PSD3 AST based on type-theoretic principles. The key insight is that D3's "join" pattern is actually a **fold over data structures**, and the specific join type is determined by the shape of the data, not special-cased in the AST.

The new design reduces 6+ join constructors to **two orthogonal iteration primitives** plus composition:

Primitive	Purpose	Data Shape
`Iterate`	Create siblings from collection	`Array a`
`Recurse`	Create hierarchy from recursive data	`Tree a`, `Cofree f a`

This transcends D3's conceptual model while remaining fully compatible with D3 as an interpreter target.

Motivation

The Revelation

While implementing `RecursiveJoin` to handle tree-structured data (using `tree-rose`'s `Cofree List a`), we discovered a deeper pattern:

```
Foldable    →  Join         (traverse a collection)
Unfoldable  →  NestedJoin   (decompose then traverse)
Recursive   →  RecursiveJoin (fold over recursive structure)
```

The "join" isn't a D3-specific operation—it's a **fold**. The template function `a -> Tree a` is constant; only the iteration pattern changes based on data shape.

Problems with Current AST

The current AST has 6+ join-related constructors:

```
| Join { ... }                -- flat iteration
| NestedJoin { ... }          -- flatten then iterate
| UpdateJoin { ... }          -- flat + GUP
| UpdateNestedJoin { ... }  -- flatten + GUP
| HierarchicalJoin { ... }  -- two-level with context
| RecursiveJoin { ... }     -- recursive traversal
```

Issues:

1. **Combinatorial explosion** - each feature (nesting, GUP, hierarchy) multiplies constructors
2. **Conflated concerns** - structure creation mixed with differential updates
3. **D3-specific terminology** - "join" requires understanding D3's mental model
4. **Non-compositional** - complex patterns require new constructors

Design Principles

1. Separate Structure from Updates

Structure creation (one-shot): "Given this data, what DOM should exist?"

Differential updates (GUP): "Given old and new states, how do we transition?"

These are orthogonal concerns. The AST should express structure; GUP is a modifier.

2. Two Iteration Modes

There are exactly two ways to iterate over data:

Flat Iteration (Siblings)

```
[a, b, c]  →  <elem/> <elem/> <elem/>
```

Create one element per datum. All elements are siblings.

Hierarchical Iteration (Nested)

```
Tree       →  <elem>
                <elem>
                  <elem/>
                </elem>
                <elem/>
              </elem>
```

Traverse recursive structure, preserving hierarchy in output.

3. Composition Over Special Cases

Complex patterns emerge from composing simple primitives:

```
-- HierarchicalJoin = nested Iterates with closure
Iterate packages \pkg ->
  Elem Group []
    [ Iterate pkg.modules \mod ->
        Elem Circle [ fill pkg.color, r mod.size ]  -- closure!
    ]
```

```
-- NestedJoin = flatMap in data, then Iterate
Iterate (packages >>= _.modules) moduleTemplate
```

4. Type-Theoretic Foundation

Concept	Type Theory	PSD3
Template	Algebra: `F a -> a`	`a -> Tree a`
Flat iteration	Functor map	`Iterate`
Recursive iteration	Catamorphism	`Recurse`
Children accessor	Coalgebra: `a -> F a`	`children :: a -> Array a`

Proposed AST

Core Data Type

```
data Tree a
  -- | A DOM element with attributes, children, and behaviors
  = Elem
      { elemType :: ElementType
      , attrs :: Array (Attribute a)
      , children :: Array (Tree a)
      , behaviors :: Array (Behavior a)
      }

  -- | Flat iteration: create sibling elements from a collection
  | Iterate
      { name :: String                -- Selection name for retrieval
      , key :: ElementType            -- Element type to create per datum
      , items :: Array a              -- The data to iterate over
      , keyFn :: a -> String          -- Identity function for diffing
      , template :: a -> Tree a       -- How to render one datum
      , gup :: Maybe (GUPSpec a)      -- Optional differential updates
      }

  -- | Recursive iteration: create nested structure from recursive data
  | Recurse
      { name :: String
      , key :: ElementType
      , root :: a                     -- Single root datum (THE tree)
      , children :: a -> Array a      -- Coalgebra: extract children
      , keyFn :: a -> String
      , template :: a -> Tree a
      , gup :: Maybe (GUPSpec a)
      }

  -- | Empty tree (for conditional rendering)
  | Empty
```

GUP Specification (Orthogonal)

```
data GUPSpec a = GUPSpec
  { enter :: Maybe (PhaseSpec a)
  , update :: Maybe (PhaseSpec a)
  , exit :: Maybe (PhaseSpec a)
  }

data PhaseSpec a = PhaseSpec
  { attrs :: Array (Attribute a)      -- Attributes for this phase
  , transition :: Maybe TransitionConfig
  }
```

Attribute Model (Unchanged)

```
data Attribute a
  = StaticAttr AttributeName AttributeValue
  | DataAttr AttributeName AttrSource (a -> AttributeValue)
  | IndexedAttr AttributeName AttrSource (a -> Int -> AttributeValue)
  | AnimatedAttr { ... }
```

Composition Patterns

Pattern 1: Simple Collection (replaces Join)

```
-- Before
Join { name: "circles", key: "circle", joinData: points, ... }

-- After
Iterate
  { name: "circles"
  , key: Circle
  , items: points
  , keyFn: _.id
  , template: \pt -> Elem { attrs: [cx pt.x, cy pt.y, r 5.0], ... }
  , gup: Nothing
  }
```

Pattern 2: Nested Data (replaces NestedJoin)

```
-- Before: NestedJoin with decompose function
NestedJoin { joinData: scenes, decompose: _.points, ... }

-- After: flatMap in the data layer
```

```
Iterate
  { items: scenes >>= _.points  -- or: concatMap _.points scenes
  , template: pointTemplate
  , ...
  }
```

Pattern 3: Parent-Child Context (replaces HierarchicalJoin)

```
-- Before: special HierarchicalJoin constructor
HierarchicalJoin
  { parentData: packages
  , getChildren: _.modules
  , template: \pkg mod -> ...  -- receives both
  }

-- After: nested Iterate with closure
Iterate
  { items: packages
  , template: \pkg ->
      Elem { elemType: Group, children:
        [ Iterate
            { items: pkg.modules
            , template: \mod ->
                -- pkg is in scope via closure!
                Elem { attrs: [fill pkg.color, r mod.size], ... }
            }
        ]
      }
  }
```

Pattern 4: Recursive Tree (replaces RecursiveJoin)

```
-- Using tree-rose's Tree (Cofree List a)
Recurse
  { name: "nodes"
  , key: Group
  , root: flareTree                        -- Single root!
  , children: Array.fromFoldable <<< tail  -- Cofree's tail
  , keyFn: \t -> (head t).name             -- Cofree's head
  , template: nodeTemplate
  , gup: Nothing
  }
```

Pattern 5: With GUP (replaces UpdateJoin)

```
Iterate
  { items: data
  , template: elementTemplate
  , gup: Just $ GUPSpec
      { enter: Just $ PhaseSpec
          { attrs: [opacity 0.0]
          , transition: Just { duration: 300, ease: EaseLinear }
          }
      , update: Just $ PhaseSpec
          { attrs: []
          , transition: Just { duration: 200, ease: EaseCubic }
          }
      , exit: Just $ PhaseSpec
          { attrs: [opacity 0.0]
          , transition: Just { duration: 200, ease: EaseLinear }
          }
      }
  }
```

Interpreter Changes

Simplified Rendering

```
renderTree :: Selection -> Tree a -> Effect (Map String Selection)

renderTree parent (Elem spec) = do
  el <- createElement spec.elemType parent
  applyAttrs el spec.attrs
  applyBehaviors el spec.behaviors
  childSels <- traverse (renderTree el) spec.children
  pure $ foldl Map.union Map.empty childSels

renderTree parent (Iterate spec) = do
  elements <- for spec.items \datum -> do
    el <- createElement spec.key parent
    childTree <- pure $ spec.template datum
    renderTree el childTree
  let sel = mkBoundSelection elements spec.items
  pure $ Map.singleton spec.name sel

renderTree parent (Recurse spec) = do
  let go node = do
        el <- createElement spec.key parent
        _ <- renderTree el (spec.template node)
        for_ (spec.children node) \child ->
          go child  -- recursive!
        pure el
  rootEl <- go spec.root
  -- collect all elements for selection...
  pure $ Map.singleton spec.name sel
```

```
renderTree _ Empty = pure Map.empty
```

GUP as Separate Pass

GUP can be handled as:

1. A modifier on the interpreter
2. A separate reconciliation pass
3. Integrated into Iterate/Recurse when `gup` is `Just`

The key insight is that GUP **diffs two Tree specifications**, not two DOM states.

Migration Path

Phase 1: New AST Module

- Create `PSD3.AST.V2` with new types
- Keep old AST for compatibility
- Build new interpreter for V2

Phase 2: Adapter Layer

- Write `convertV1toV2 :: AST.Tree a -> AST.V2.Tree a`
- Existing code continues to work
- New code can use V2 directly

Phase 3: Migrate Demos

- Update demos one by one to V2
- Validate visual parity
- Remove V1 constructs as demos migrate

Phase 4: Remove V1

- Delete old AST constructors
- Update all imports
- Remove adapter layer

Benefits

Conceptual Clarity

- No D3-specific "join" terminology
- Based on universal fold/iteration concepts
- Clear separation of concerns

Reduced Surface Area

- 3 constructors vs 8+

- Fewer special cases to learn
- Easier to reason about

Compositionality

- Complex patterns from simple primitives
- Closures handle context naturally
- No need for special "hierarchical" modes

Type-Theoretic Foundation

- Grounded in algebras and coalgebras
- Iteration mode follows from data shape
- Extensible to new data structures

Easier Testing

- Smaller AST = fewer cases to test
- Composition is easier to verify
- Interpreters are simpler

Open Questions

1. Key Element Wrapping

Current joins create a wrapper element (e.g., `<g>`) for each datum. Should this be:

- Implicit in Iterate/Recurse (current behavior)
- Explicit in the template
- Configurable

Recommendation: Explicit in template for clarity.

2. Selection Retrieval

The `name` field enables `Map String Selection` retrieval. Is this sufficient for:

- Coordinated highlighting
- Post-render updates
- Animation targeting

Recommendation: Yes, with accessor functions.

3. Conditional Rendering

How to handle "render A or B based on predicate"?

Options:

- Use `Empty` and `if/then/else` in template
- Add `Conditional` constructor
- Pattern match in template function

Recommendation: Templates are functions; use normal branching.

4. Behaviors Location

Currently behaviors can be on any Tree node. Should they only be on Elem?

Recommendation: Only on Elem—behaviors attach to DOM elements.

5. Transition Semantics

Are transitions part of GUP or a separate `Animate` constructor?

Recommendation: Part of GUP. Transitions describe phase changes.

Conclusion

This redesign reconceptualizes PSD3's AST around type-theoretic principles:

- **Iterate**: functor map over collections (siblings)
- **Recurse**: catamorphism over recursive structures (hierarchy)
- **GUP**: orthogonal differential update specification
- **Composition**: complex patterns from simple primitives

The result is a smaller, more principled, more composable system that transcends D3's conceptual model while remaining compatible with D3 as an interpreter target.

The migration will require updating all demos and showcases, but the payoff is a foundation that is:

- Easier to teach
- Easier to extend
- Grounded in universal concepts
- Not tied to D3's historical baggage

Appendix: Type-Theoretic Grounding

Algebras and Folds

An **F-algebra** is a pair `(A, alg)` where:

- `A` is a carrier type (our DOM specification)
- `alg :: F A -> A` is the algebra morphism

For rose trees, `F = TreeF a` where `data TreeF a r = NodeF a [r]`.

Our template is an algebra: given the node's data and rendered children, produce the DOM for this node.

Coalgebras and Unfolds

A **coalgebra** is `coalg :: A -> F A`.

Our `children :: a -> Array a` is a coalgebra—it shows how to decompose a node into its children.

The Fold

A **catamorphism** (fold) uses an algebra to consume recursive structure:

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

Our `Recurse` implements exactly this pattern, where the algebra is the template function.

Why This Matters

Grounding in these concepts means:

1. We can leverage decades of research on recursion schemes
2. The design is provably compositional
3. New data structures (graphs, DAGs) can be added by defining their base functors
4. Optimizations (fusion, deforestation) become applicable