Algebraic Structures in PSD3v2

This document explores the latent algebraic structures in the library and whether we should expose them as typeclass instances.

1. Selection as a Monoid? (Almost!)

The Monoid-ish Feel

You're right that `Selection` has a monoid-ish quality, but it's **not quite** a Monoid:

```
-- What we have:
merge :: Selection SBound Element datum
      -> Selection SBound Element datum
      -> Effect (Selection SBound Element datum)

-- What Monoid needs:
(<>) :: Selection SBound Element datum
     -> Selection SBound Element datum
     -> Selection SBound Element datum  -- Pure!

mempty :: Selection SBound Element datum  -- No datum available!
```

Problems:

1. **Effect wrapper**: `merge` lives in `Effect`, not pure
2. **mempty is impossible**: We can't create an empty `SBound` selection without a `Document`
3. **State restrictions**: Only `SBound` selections can merge (phantom types prevent merging `SPending` with `SExiting`)

What it ACTUALLY is:

A Semigroup in the Effect monad with state constraints!

```
-- This WOULD work:
instance Semigroup (EffectfulBoundSelection datum) where
  append (EffectfulBoundSelection sel1) (EffectfulBoundSelection sel2) =
    EffectfulBoundSelection (merge sel1 sel2)

-- But we'd need a newtype wrapper:
newtype EffectfulBoundSelection datum =
  EffectfulBoundSelection (Effect (Selection SBound Element datum))
```

Verdict: Don't add Monoid instance. The Effect wrapper and phantom type restrictions make it unnatural. The explicit `merge` function is clearer.

2. Selection as a Functor? (YES!)

The Opportunity

Selections are **absolutely** functors over their datum type:

```
instance Functor (Selection state parent) where
  map :: forall a b. (a -> b)
      -> Selection state parent a
      -> Selection state parent b
  map f (Selection impl) = Selection (mapImpl f impl)
    where
      mapImpl :: (a -> b) -> SelectionImpl parent a -> SelectionImpl
parent b
      mapImpl f (BoundSelection r) = BoundSelection
        { elements: r.elements
        , data: map f r.data
        , indices: r.indices
        , document: r.document
        }
      mapImpl f (PendingSelection r) = PendingSelection
        { parentElements: r.parentElements
        , pendingData: map f r.pendingData
        , indices: r.indices
        , document: r.document
        }
      mapImpl f (ExitingSelection r) = ExitingSelection
        { elements: r.elements
        , data: map f r.data
        , document: r.document
        }
      mapImpl f (EmptySelection r) = EmptySelection r  -- No data to map
```

Use Cases

```
-- Transform bound data without touching DOM
prices :: Selection SBound Element Number
pricesInEuros :: Selection SBound Element Number
pricesInEuros = map (_ * 0.85) prices

-- Extract field from records
people :: Selection SBound Element { name :: String, age :: Int }
ages :: Selection SBound Element Int
ages = map _.age people

-- Compose with other functors
maybeSelection :: Maybe (Selection SBound Element Int)
doubled :: Maybe (Selection SBound Element Int)
doubled = map (map (_ * 2)) maybeSelection
```

Verdict: **YES, add Functor instance!** This is a natural fit and very useful.

---

3. Tree as a Free Monad? (Fascinating!)

The Structure

`Tree` is suspiciously monad-like:

```
data Tree datum
  = Node (TreeNode datum)
  | Join { template :: datum -> Tree datum, ... }
  | NestedJoin { template :: datum -> Tree datum, ... }

-- It has:
-- - Pure values: Node
-- - Bind-like structure: Join/NestedJoin build trees from functions
```

But it's not *quite* a monad because:

1. `Join` and `NestedJoin` have extra structure (name, key, data)
2. The recursion is controlled (not arbitrary bind)

What it IS:

A domain-specific tree structure that embeds functions

It's closer to a **Cofree Comonad** or **Rose Tree** than a Monad:

```
-- Rose Tree (similar structure):
data Tree a = Node a (Array (Tree a))

-- Our Tree (with data binding points):
data Tree datum
  = Node { attrs :: Array (Attribute datum), children :: Array (Tree
datum) }
  | Join { template :: datum -> Tree datum, ... }
```

Verdict: Don't force it into Monad. The structure is domain-specific and the types don't align naturally.

---

4. Attribute as a Contravariant Functor (Profunctor!)

The Insight

Attributes consume data but produce side effects:

```
data Attribute datum
  = StaticAttr AttributeName AttributeValue
  | DataAttr AttributeName (datum -> AttributeValue)
  | IndexedAttr AttributeName (datum -> Int -> AttributeValue)
```

This is **Contravariant** in `datum`:

```
instance Contravariant Attribute where
  cmap :: forall a b. (b -> a) -> Attribute a -> Attribute b
  cmap f (StaticAttr name val) = StaticAttr name val
  cmap f (DataAttr name g) = DataAttr name (g <<< f)
  cmap f (IndexedAttr name g) = IndexedAttr name (\b i -> g (f b) i)
```

Use Cases

```
-- Adapt attributes to work on different data types
ageAttr :: Attribute Int
ageAttr = DataAttr "data-age" (show >>> StringValue)

personAgeAttr :: Attribute { age :: Int, name :: String }
personAgeAttr = cmap _.age ageAttr

-- Compose with other contravariant functors
type Setter a = a -> Effect Unit

combinedEffect :: Attribute Int -> Setter { age :: Int } -> Setter { age
:: Int }
combinedEffect attr setter = cmap _.age setter
```

Verdict: **YES, add Contravariant instance!** This is mathematically sound and practically useful.

---

5. JoinResult as a Bifunctor or Trifunctor

The Structure

```
data JoinResult sel parent datum = JoinResult
  { enter  :: sel SPending parent datum
  , update :: sel SBound Element datum
  , exit   :: sel SExiting Element datum
  }
```

This varies in **three** type parameters:

- `sel` (the selection type constructor)

- `parent` (parent element type)
- `datum` (bound data type)

Possible Instances

```
-- Functor over datum (most useful):
instance Functor (JoinResult sel parent) where
  map f (JoinResult { enter, update, exit }) = JoinResult
    { enter: map f enter   -- Requires Functor (sel SPending parent)
    , update: map f update -- Requires Functor (sel SBound Element)
    , exit: map f exit     -- Requires Functor (sel SExiting Element)
    }
```

Problem: This requires `sel` to be functorial in all phantom state types, which is true for `Selection` but might not be true for other interpreters.

Verdict: Maybe useful, but only if we add Functor to Selection first. Consider it once Selection has Functor.

---

6. Monoidal Structure on Tree Construction

The Pattern

Tree building has an **Applicative-like** composition:

```
-- Current API:
parent `withChild` child
parent `withChildren` [child1, child2, child3]

-- Applicative-like:
-- pure = elem
-- <*> = withChild (kind of)
```

But the types don't align naturally:

```
-- What we'd need:
class TreeBuilder t where
  pure :: ElementType -> Array (Attribute datum) -> t datum
  (<*>) :: t datum -> t datum -> t datum

-- withChild is associative (tree structure)
-- But no sensible "mempty" (no empty tree that makes sense)
```

Verdict: This is more of a **Builder pattern** than a Monad/Applicative. The current API is fine.

Summary: What to Add

Definite YES:

1. **Functor for Selection** - Natural, useful, mathematically sound
2. **Contravariant for Attribute** - Elegant, enables attribute reuse

Maybe Later:

3. **Functor for JoinResult** - Useful but requires Selection Functor first
4. **Semigroup for EffectfulBoundSelection** - Possible but adds complexity

No:

5. **Monoid for Selection** - Effect and phantom types make it unnatural
6. **Monad for Tree** - Structure is domain-specific, not general recursion
7. **Applicative for Tree building** - No sensible identity element

---

Implementation Sketch

```
-- Selection.Types.purs

instance Functor (Selection state parent) where
  map f (Selection impl) = Selection (mapImpl f impl)
    where
      mapImpl :: (a -> b) -> SelectionImpl parent a -> SelectionImpl
parent b
      mapImpl f (EmptySelection r) = EmptySelection r
      mapImpl f (BoundSelection r) = BoundSelection
        { elements: r.elements
        , data: map f r.data
        , indices: r.indices
        , document: r.document
        }
      mapImpl f (PendingSelection r) = PendingSelection
        { parentElements: r.parentElements
        , pendingData: map f r.pendingData
        , indices: r.indices
        , document: r.document
        }
      mapImpl f (ExitingSelection r) = ExitingSelection
        { elements: r.elements
        , data: map f r.data
        , document: r.document
        }

-- Attribute.Types.purs

instance Contravariant Attribute where
  cmap f (StaticAttr name val) = StaticAttr name val
  cmap f (DataAttr name g) = DataAttr name (g <<< f)
  cmap f (IndexedAttr name g) = IndexedAttr name (\b i -> g (f b) i)
```

Conceptual Benefits

Functor for Selection

- Enables standard `map` over data
- Composes with `traverse`, `sequence`, etc.
- Makes data transformations explicit
- No runtime overhead (pure function application)

Contravariant for Attribute

- Enables attribute reuse across different data types
- Makes data flow explicit (consume data, produce effects)
- Composes with other contravariant functors
- Clarifies the "consumer" nature of attributes

Why Not Others

- Monoid/Semigroup: Effect wrapper breaks purity laws
- Monad: Tree structure is too specific
- Applicative: No sensible identity/zero

Open Questions

1. Should we also add `Bifunctor` instances for anything?
2. Are there **Profunctor** opportunities? (functions `a -> b` embedded in structures)
3. Could `renderTree` itself have algebraic structure? (It's a catamorphism!)
4. Is there a **Traversable** instance for Tree? (Walk and accumulate effects)

These are worth exploring once the basic Functor/Contravariant instances are in place.