PSD3 Selection

Table of Contents

About This Documentation

PSD3 Selection is a type-safe, finally tagless D3-style visualization library for PureScript. This documentation covers the core concepts, how-to guides, and reference material.

Getting Started

Chapter 1. Your First Selection

In this tutorial, you'll create a simple visualization using PSD3. By the end, you'll understand the core pattern: build a **tree AST** that describes your visualization, then interpret it to render.

1.1. Prerequisites

- PureScript and Spago installed
- Basic familiarity with PureScript syntax
- A project with `psd3-selection` installed:

```
spago install psd3-selection
```

1.2. The Core Pattern

PSD3 is different from D3.js. Instead of imperative method chaining:

```
// D3.js (imperative)
d3.select("body")
  .append("svg")
  .attr("width", 400)
  .attr("height", 300)
  .append("circle")
  .attr("cx", 200)
  .attr("cy", 150)
  .attr("r", 50)
  .attr("fill", "steelblue");
```

You build a **tree** that describes what you want:

```
-- PSD3 (declarative)
myViz :: A.AST Unit
myViz =
  A.elem SVG
    [ width (num 400.0)
    , height (num 300.0)
    ]
    `A.withChild`
      A.elem Circle
        [ cx (num 200.0)
        , cy (num 150.0)
        , r (num 50.0)
        , fill (text "steelblue")
        ]
```

Then you render that tree to the DOM. The tree is **data** - you can inspect it, transform it, test it.

1.3. Step 1: Imports

Create a file `src/Main.purs`:

```
module Main where

import Prelude

import Effect (Effect)
import Effect.Console (log)

import PSD3.AST as A
import PSD3.AST (ElementType(..))
import PSD3.Expr.Friendly (num, text, width, height, cx, cy, r, fill)
import PSD3.Render (runD3, select, renderTree)
```

The key imports are:

- `PSD3.AST` - Tree building functions (`elem`, `withChild`, etc.)
- `PSD3.AST (ElementType(..))` - Element types (`SVG`, `Circle`, `Rect`, etc.)
- `PSD3.Expr.Friendly` - Attribute helpers (`num`, `text`, `width`, `cx`, etc.)
- `PSD3.Render` - Rendering functions (`runD3`, `select`, `renderTree`)

1.4. Step 2: Build the Tree

```
myViz :: A.AST Unit
myViz =
  A.elem SVG
    [ width (num 400.0)
    , height (num 300.0)
    ]
    `A.withChild`
      A.elem Circle
        [ cx (num 200.0)
        , cy (num 150.0)
        , r (num 50.0)
        , fill (text "steelblue")
        ]
```

Let's break this down:

`A.AST Unit`
: The tree type. `Unit` means we're not binding any data yet.

`A.elem SVG [⋯]`

Create an SVG element with attributes.

`width (num 400.0)`

Set width to 400. `num` wraps a number in the expression system.

`` \`A.withChild\` ``

Add a child element. The backticks make it infix.

`fill (text "steelblue")`

String attributes use `text` instead of `num`.

1.5. Step 3: Render to the DOM

```
main :: Effect Unit
main = void $ runD3 do
  container <- select "body"
  renderTree container myViz
  pure unit
```

`runD3`

Runs a D3 computation in the Effect monad.

`select "body"`

Select the body element as our container.

`renderTree container myViz`

Render the AST into the container.

1.6. Complete Example

```
module Main where

import Prelude

import Effect (Effect)

import PSD3.AST as A
import PSD3.AST (ElementType(..))
import PSD3.Expr.Friendly (num, text, width, height, cx, cy, r, fill)
import PSD3.Render (runD3, select, renderTree)

myViz :: A.AST Unit
myViz =
  A.elem SVG
    [ width (num 400.0)
```

```
    , height (num 300.0)
    ]
    `A.withChild`
      A.elem Circle
        [ cx (num 200.0)
        , cy (num 150.0)
        , r (num 50.0)
        , fill (text "steelblue")
        ]

main :: Effect Unit
main = void $ runD3 do
  container <- select "body"
  renderTree container myViz
  pure unit
```

1.7. Adding More Elements

Use `withChildren` to add multiple children:

```
myViz :: A.AST Unit
myViz =
  A.elem SVG
    [ width (num 400.0)
    , height (num 300.0)
    ]
    `A.withChildren`
      [ A.elem Circle
          [ cx (num 100.0), cy (num 150.0), r (num 40.0)
          , fill (text "steelblue")
          ]
      , A.elem Circle
          [ cx (num 200.0), cy (num 150.0), r (num 40.0)
          , fill (text "coral")
          ]
      , A.elem Circle
          [ cx (num 300.0), cy (num 150.0), r (num 40.0)
          , fill (text "seagreen")
          ]
      ]
```

1.8. Nesting Elements

Trees can be nested arbitrarily deep:

```
import PSD3.AST (ElementType(..))  -- includes Group
import PSD3.Expr.Friendly (transform)
```

```
myViz :: A.AST Unit
myViz =
  A.elem SVG
    [ width (num 400.0), height (num 300.0) ]
    `A.withChild`
      A.elem Group
        [ transform (text "translate(200, 150)") ]
        `A.withChildren`
          [ A.elem Circle [ r (num 50.0), fill (text "steelblue") ]
          , A.elem Circle [ r (num 30.0), fill (text "white") ]
          ]
```

1.9. Named Elements

Use `A.named` instead of `A.elem` when you need to reference an element later:

```
myViz :: A.AST Unit
myViz =
  A.named SVG "chart"
    [ width (num 400.0), height (num 300.0) ]
    `A.withChild`
      A.named Circle "mainCircle"
        [ cx (num 200.0), cy (num 150.0), r (num 50.0)
        , fill (text "steelblue")
        ]
```

Named selections can be retrieved after rendering for updates or attaching behaviors.

1.10. What You've Learned

1. **Build trees, not commands** - PSD3 visualizations are data structures
2. `A.elem` creates anonymous elements, `A.named` creates retrievable ones
3. `withChild` and `withChildren` nest elements
4. `num` and `text` wrap values for the attribute system
5. `runD3` executes the rendering

1.11. Next Steps

- Building a Bar Chart - Data binding with `joinData`
- Coordinated Highlighting - Interactive behaviors
- The Selection AST - Deep dive into the tree structure

Chapter 2. Building a Bar Chart

This tutorial is coming soon. It will cover:

- Binding data with `joinData`
- Using `from` for data-driven attributes
- Scales and axes
- Adding interactivity

2.1. Preview

```
barChart :: Array Number -> A.AST Number
barChart data' =
  A.elem SVG
    [ width (num 500.0)
    , height (num 300.0)
    ]
    `A.withChild`
      A.joinData "bars" "rect" data' \d ->
        A.elem Rect
          [ from "x" (\_ i -> toNumber i * 25.0)
          , from "y" (\val _ -> 300.0 - val)
          , width (num 20.0)
          , from "height" (\val _ -> val)
          , fill (text "steelblue")
          ]
```

2.2. Next Steps

- Your First Selection - Start here if you haven't
- Data Joins Deep Dive - Advanced join patterns

Chapter 3. From D3.js to PSD3

This guide is coming soon. It will cover:

- Key conceptual differences
- Mapping D3 patterns to PSD3
- Common gotchas
- When to use the escape hatches

3.1. Quick Comparison

D3.js	PSD3
Imperative method chaining	Declarative tree building
Mutable selections	Immutable AST
Runtime errors	Compile-time type safety
Single interpreter (DOM)	Multiple interpreters (DOM, String, Mermaid)

3.2. The Big Idea

In D3, you **do** things to the DOM:

```
d3.select("body").append("svg").attr("width", 400)...
```

In PSD3, you **describe** what you want:

```
A.elem SVG [ width (num 400.0) ] `A.withChild` ...
```

Then an interpreter renders it.

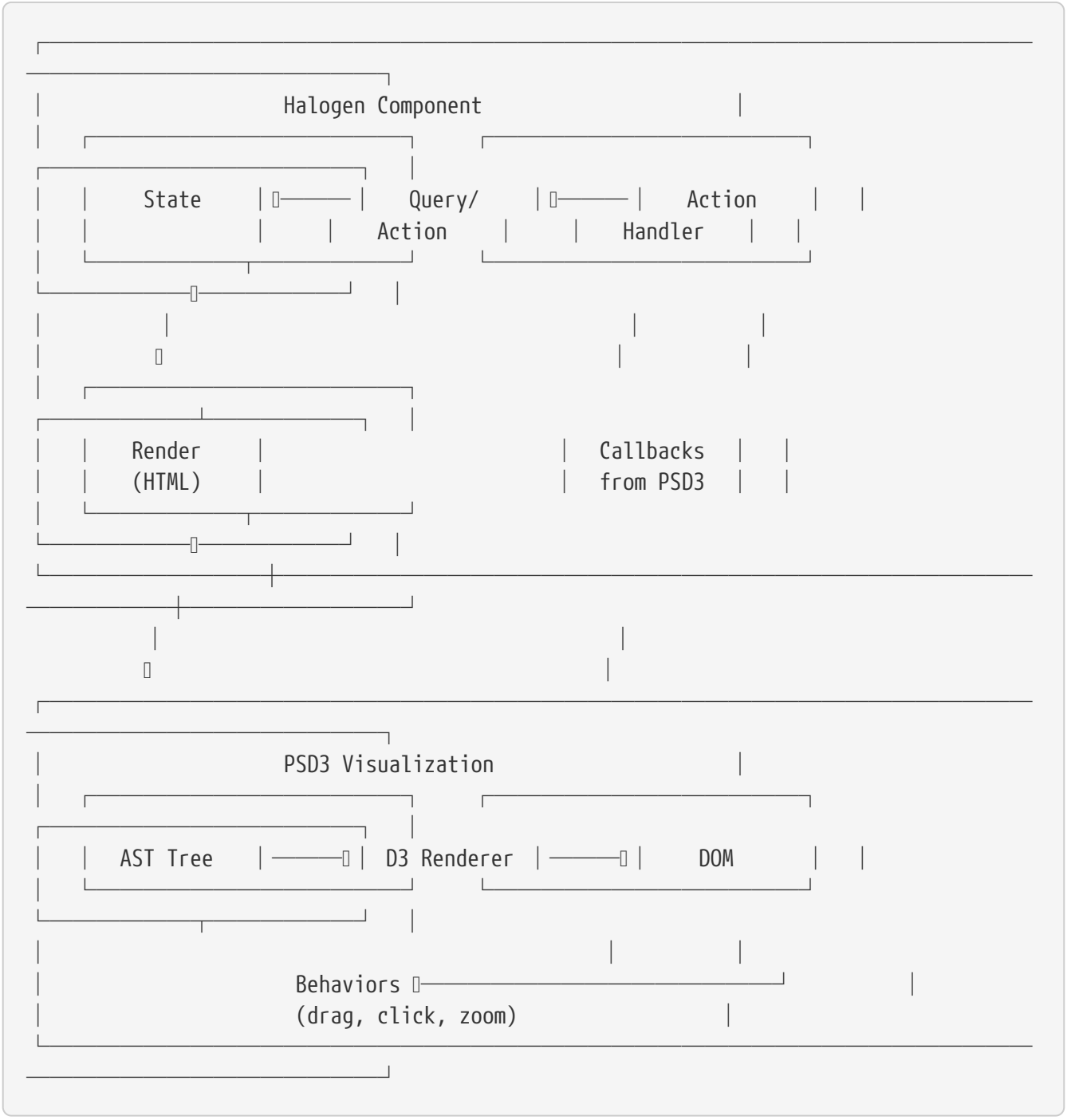
3.3. Next Steps

- Your First Selection
- Finally Tagless Architecture

Chapter 4. Web App Architecture with PSD3

Building a web application with a data visualization component requires thinking about how user interactions flow between your UI framework and your visualizations. This guide covers the recommended architecture.

4.1. The Core Pattern



4.2. Key Concepts

Halogen owns the state

Your Halogen component holds the application state. The visualization is a view of that state.

PSD3 renders declaratively

You build an AST from your state, PSD3 renders it. When state changes, you re-render.

Events flow back via callbacks

Clicks, drags, and hovers in the visualization trigger callbacks that become Halogen actions.

4.3. Basic Structure

```
module MyApp.Visualization where

import Halogen as H
import Halogen.HTML as HH
import PSD3 as P

type State =
  { data :: Array DataPoint
  , selectedId :: Maybe String
  }

data Action
  = Initialize
  | DataPointClicked String
  | DataPointDragged String { x :: Number, y :: Number }

component :: forall q i o m. MonadEffect m => H.Component q i o m
component = H.mkComponent
  { initialState: \_ -> { data: [], selectedId: Nothing }
  , render
  , eval: H.mkEval $ H.defaultEval
      { handleAction = handleAction
      , initialize = Just Initialize
      }
  }

render :: State -> H.ComponentHTML Action () m
render state =
  HH.div
    [ HP.ref (H.RefLabel "viz-container") ]
    []  -- PSD3 will render into this div

handleAction :: forall o m. MonadEffect m => Action -> H.HalogenM State Action () o m
Unit
handleAction = case _ of
  Initialize -> do
    -- Get the container element and render initial visualization
    mContainer <- H.getRef (H.RefLabel "viz-container")
    for_ mContainer \container -> do
      state <- H.get
      liftEffect $ renderVisualization container state handleVizEvent
```

```
  DataPointClicked id -> do
    H.modify_ _ { selectedId = Just id }
    -- Re-render visualization with new selection
    reRender

  DataPointDragged id pos -> do
    H.modify_ \s -> s { data = updatePosition id pos s.data }
    reRender
```

4.4. The Callback Bridge

The key to integrating PSD3 with Halogen is the callback bridge:

```
-- In your visualization module
renderVisualization
  :: Element
  -> State
  -> (VizEvent -> Effect Unit)  -- Callback to Halogen
  -> Effect Unit
renderVisualization container state onEvent = void $ runD3 do
  svg <- select container
  let ast = buildVisualization state onEvent
  renderTree svg ast

buildVisualization :: State -> (VizEvent -> Effect Unit) -> P.AST DataPoint
buildVisualization state onEvent =
  P.elem SVG [ width (num 800.0), height (num 600.0) ]
    `P.withChild`
      P.joinData "points" "circle" state.data \d ->
        P.elem Circle
          [ cx (from _.x)
          , cy (from _.y)
          , r (num 5.0)
          ]
          `P.withBehaviors`
            [ onClick \datum -> onEvent (PointClicked datum.id)
            , onDrag \datum pos -> onEvent (PointDragged datum.id pos)
            ]
```

4.5. Next Steps

- Halogen Events from PSD3 - Detailed event handling
- PSD3 without Halogen - Simpler setups

How-To Guides

Chapter 5. PSD3 in a Simple Web Page

Not every visualization needs a full application framework. This guide shows how to use PSD3 in a simple HTML page with minimal setup.

5.1. When to Use This Approach

- Quick prototypes and experiments
- Static visualizations with no complex state
- Embedding a visualization in an existing non-PureScript page
- Learning PSD3 without Halogen complexity

5.2. Basic Setup

5.2.1. 1. Create Your HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>My Visualization</title>
  <style>
    #chart { width: 800px; height: 600px; }
  </style>
</head>
<body>
  <div id="chart"></div>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <script src="./my-viz.js"></script>
</body>
</html>
```

5.2.2. 2. Write Your PureScript

```
module Main where

import Prelude
import Effect (Effect)
import PSD3.AST as A
import PSD3.AST (ElementType(..))
import PSD3.Expr.Friendly (num, text, width, height, cx, cy, r, fill)
import PSD3.Render (runD3, select, renderTree)

main :: Effect Unit
main = void $ runD3 do
  container <- select "#chart"
```

```
  renderTree container myVisualization

myVisualization :: A.AST Unit
myVisualization =
  A.elem SVG
    [ width (num 800.0)
    , height (num 600.0)
    ]
    `A.withChild`
      A.elem Circle
        [ cx (num 400.0)
        , cy (num 300.0)
        , r (num 100.0)
        , fill (text "steelblue")
        ]
```

5.2.3. 3. Bundle and Run

```
# Build PureScript
spago build

# Bundle for browser
spago bundle --module Main --outfile my-viz.js

# Open in browser
open index.html
```

5.3. Adding Interactivity

For simple interactivity, use behaviors directly:

```
import PSD3.Internal.Behavior.Types (onClick, onHover)
import Effect.Console (log)

myVisualization :: A.AST Unit
myVisualization =
  A.elem SVG [ width (num 800.0), height (num 600.0) ]
    `A.withChild`
      A.elem Circle
        [ cx (num 400.0), cy (num 300.0), r (num 100.0)
        , fill (text "steelblue")
        ]
        `A.withBehaviors`
          [ onClick \_ -> log "Circle clicked!"
          , onHover
              { enter: [{ attr: "fill", value: "coral" }]
              , leave: [{ attr: "fill", value: "steelblue" }]
              }
```

```
        ]
```

5.4. Managing State with Effect.Ref

For simple state without a framework, use `Effect.Ref`:

```
import Effect.Ref as Ref

main :: Effect Unit
main = do
  -- Mutable state
  countRef <- Ref.new 0

  void $ runD3 do
    container <- select "#chart"

    let viz count =
          A.elem SVG [ width (num 800.0), height (num 600.0) ]
            `A.withChild`
              A.elem Text
                [ x (num 400.0), y (num 300.0)
                , textContent (text $ "Count: " <> show count)
                ]
                `A.withBehaviors`
                  [ onClick \_ -> do
                      newCount <- Ref.modify (_ + 1) countRef
                      -- Re-render with new count
                      reRender container (viz newCount)
                  ]

    count <- liftEffect $ Ref.read countRef
    renderTree container (viz count)
```

5.5. Loading External Data

```
import Affjax.Web as Ajax
import Affjax.ResponseFormat as RF

main :: Effect Unit
main = launchAff_ do
  result <- Ajax.get RF.json "/data/points.json"
  case result of
    Left err -> log $ "Failed to load data: " <> printError err
    Right response -> do
      let points = decodePoints response.body
      liftEffect $ renderVisualization points
```

```
renderVisualization :: Array Point -> Effect Unit
renderVisualization points = void $ runD3 do
  container <- select "#chart"
  renderTree container (scatterplot points)
```

5.6. When to Graduate to Halogen

Consider moving to Halogen when you need:

- Complex application state
- Multiple interacting components
- Routing
- Server communication with loading states
- Undo/redo functionality

5.7. Next Steps

- Your First Selection - PSD3 basics
- Simple CSS Hover Effects - Lightweight interactivity
- Web App Architecture - When you're ready for more

Chapter 6. Halogen Events from PSD3 Visualizations

When building a Halogen application with PSD3 visualizations, you need to bridge events from the D3 world back to Halogen's action system. This guide shows you how.

6.1. The Challenge

PSD3 visualizations live in the DOM, managed by D3. Halogen manages your application state. When a user clicks or drags something in the visualization, you need to:

1. Capture the event in the D3 world
2. Extract relevant data (which element, position, etc.)
3. Trigger a Halogen action
4. Update state and re-render

6.2. Solution: Callback Functions

Pass a callback function from Halogen into your visualization code:

```
-- Define your visualization events
data VizEvent
  = NodeClicked NodeId
  | NodeDragged NodeId Position
  | BackgroundClicked
  | ZoomChanged ZoomTransform

-- In your Halogen component
handleAction = case _ of
  Initialize -> do
    mContainer <- H.getRef vizRef
    emitter <- H.emitAction  -- Get the action emitter
    for_ mContainer \el ->
      liftEffect $ renderViz el state (emitter <<< VizEventReceived)

  VizEventReceived event -> case event of
    NodeClicked id -> H.modify_ _ { selected = Just id }
    NodeDragged id pos -> updateNodePosition id pos
    -- etc.
```

6.3. Click Events

```
import PSD3.Internal.Behavior.Types (onClick, onClickWithDatum)
```

```
-- Simple click (no datum needed)
P.elem Circle [...]
  `P.withBehaviors`
    [ onClick \_ -> onEvent BackgroundClicked ]

-- Click with datum access
P.joinData "nodes" "g" nodes \node ->
  P.elem Group [...]
    `P.withBehaviors`
      [ onClickWithDatum \d -> onEvent (NodeClicked d.id) ]
```

6.4. Drag Events

```
import PSD3.Internal.Behavior.Types (Behavior(..), DragBehavior(..))

-- Full drag with start/drag/end
P.elem Circle [...]
  `P.withBehaviors`
    [ Drag $ CustomDrag
        { onStart: \d pos -> onEvent (DragStarted d.id pos)
        , onDrag: \d pos -> onEvent (NodeDragged d.id pos)
        , onEnd: \d pos -> onEvent (DragEnded d.id pos)
        }
    ]

-- Simple drag (just position updates)
P.elem Circle [...]
  `P.withBehaviors`
    [ Drag $ SimpleDrag ]  -- Updates __data__.x and __data__.y directly
```

6.5. Pattern: Effect.Ref for Mutable State

For high-frequency events like dragging, you might not want to trigger Halogen actions on every pixel move. Use `Effect.Ref` for intermediate state:

```
renderViz container state onEvent = do
  -- Mutable ref for drag state (doesn't trigger Halogen re-render)
  dragRef <- Ref.new Nothing

  void $ runD3 do
    -- ... build viz ...
    P.elem Circle [...]
      `P.withBehaviors`
        [ Drag $ CustomDrag
            { onStart: \d pos -> Ref.write (Just { id: d.id, start: pos }) dragRef
            , onDrag: \d pos -> do
                -- Update DOM directly for smooth dragging
```

```
                updateCirclePosition d.id pos
            , onEnd: \d pos -> do
                -- Only notify Halogen when drag ends
                mDrag <- Ref.read dragRef
                for_ mDrag \drag ->
                  onEvent (NodeDragComplete drag.id pos)
                Ref.write Nothing dragRef
            }
        ]
```

6.6. Pattern: Debouncing Events

For events that fire rapidly (zooming, brushing), debounce before triggering Halogen:

```
import Effect.Timer (setTimeout)

mkDebouncedEmitter :: Int -> (a -> Effect Unit) -> Effect (a -> Effect Unit)
mkDebouncedEmitter delayMs emit = do
  timerRef <- Ref.new Nothing
  pure \value -> do
    -- Cancel pending timer
    mTimer <- Ref.read timerRef
    for_ mTimer clearTimeout
    -- Set new timer
    timer <- setTimeout delayMs (emit value)
    Ref.write (Just timer) timerRef
```

6.7. Complete Example

See the Force Explorer showcase for a complete working example of Halogen + PSD3 integration with drag, zoom, and click events.

6.8. Next Steps

- Web App Architecture - The big picture
- Making Visualizations Zoomable - Zoom behavior
- PSD3 without Halogen - Simpler alternative

Chapter 7. PSD3 in a React Application

PSD3 is framework-agnostic: the same visualization code works whether you're using React, Halogen, or vanilla JavaScript. This guide shows how to integrate PSD3 with `purescript-react-basic-hooks`.

7.1. The Key Insight

PSD3 doesn't care about your UI framework. It just needs:

1. A DOM element to render into
2. A trigger to render (mount, prop change, etc.)

The framework provides the container and lifecycle; PSD3 handles the visualization.

7.2. The Pattern

```
mkMyVisualization :: Component { data :: Array Number }
mkMyVisualization = do
  -- 1. Generate unique container ID (once per component instance)
  containerId <- useContainerId

  component "MyVisualization" \props -> React.do
    -- 2. Render visualization when component mounts
    useEffectOnce do
      void $ runD3 do
        let selector = "#" <> containerId
        clear selector
        container <- select selector
        renderTree container (myVisualizationAST props.data)
      pure mempty

    -- 3. Return container div with our unique ID
    pure $ R.div
      { id: containerId
      , style: R.css { width: "500px", height: "300px" }
      }
```

7.3. Container ID Helper

Each component instance needs a unique container ID to avoid conflicts when multiple visualizations are on the same page.

src/PSD3/React/Hooks.purs

```
module PSD3.React.Hooks (useContainerId) where
```

```
import Effect (Effect)

foreign import generateContainerIdImpl :: Effect String

useContainerId :: Effect String
useContainerId = generateContainerIdImpl
```

src/PSD3/React/Hooks.js

```
let counter = 0;

export const generateContainerIdImpl = () => `psd3-container-${counter++}`;
```

7.4. Complete Bar Chart Example

Here's a complete example showing a data-bound bar chart:

```
module PSD3.React.Example where

import Prelude

import Data.Int (toNumber)
import Effect (Effect)
import PSD3.AST as A
import PSD3.AST (ElementType(..))
import PSD3.Expr.Friendly (num, text, computed, computedStr, fromWithIndex,
fromStrWithIndex)
import PSD3.React.Hooks (useContainerId)
import PSD3.Render (runD3, select, renderTree, clear)
import React.Basic.DOM as R
import React.Basic.Hooks (Component, component, useEffectOnce)
import React.Basic.Hooks as React

-- | Bar chart component with data binding
mkBarChart :: Component { chartData :: Array Number }
mkBarChart = do
  containerId <- useContainerId

  component "BarChart" \props -> React.do
    useEffectOnce do
      void $ runD3 do
        let selector = "#" <> containerId
        clear selector
        container <- select selector
        renderTree container (barChartAST props.chartData)
      pure mempty

    pure $ R.div
```

```
      { id: containerId
      , style: R.css { width: "500px", height: "300px" }
      }

-- | The visualization AST - identical to what you'd write for Halogen
barChartAST :: Array Number -> A.AST Number
barChartAST dataset =
  A.elem SVG
    [ computed "width" (num 500.0)
    , computed "height" (num 300.0)
    , computedStr "style" (text "background: #f5f5f5")
    ]
    `A.withChild`
      A.joinData "bars" "rect" dataset \_ ->
        A.elem Rect
          -- x position depends on index (bar number)
          [ fromWithIndex "x" (\_ i -> toNumber i * 60.0 + 20.0)
          -- y position depends on value (grows upward)
          , fromWithIndex "y" (\val _ -> 280.0 - val * 2.5)
          -- fixed width
          , computed "width" (num 50.0)
          -- height depends on value
          , fromWithIndex "height" (\val _ -> val * 2.5)
          -- color depends on value threshold
          , fromStrWithIndex "fill" (\val _ ->
              if val > 75.0 then "steelblue"
              else if val > 50.0 then "cornflowerblue"
              else "lightsteelblue")
          ]
```

7.5. Using DOM Elements Directly

If you prefer using React refs instead of CSS selectors, PSD3 provides `selectElement`:

```
import PSD3.Render (runD3, selectElement, renderTree)
import React.Basic.Hooks (useRef)
import Web.HTML.HTMLElement (toElement)
import Data.Nullable (toMaybe)

mkChartWithRef :: Component { data :: Array Number }
mkChartWithRef = do
  component "ChartWithRef" \props -> React.do
    containerRef <- useRef null

    useEffectOnce do
      for_ (toMaybe =<< toMaybe containerRef.current) \htmlEl -> do
        let element = toElement htmlEl
        void $ runD3 do
          container <- selectElement element
```

```
          renderTree container (myChartAST props.data)
      pure mempty

    pure $ R.div { ref: containerRef }
```

7.6. Responding to Prop Changes

To re-render when props change, use `useEffect` with a dependency:

```
mkDynamicChart :: Component { chartData :: Array Number }
mkDynamicChart = do
  containerId <- useContainerId

  component "DynamicChart" \props -> React.do
    -- Re-render when chartData changes
    useEffect props.chartData do
      void $ runD3 do
        let selector = "#" <> containerId
        clear selector
        container <- select selector
        renderTree container (barChartAST props.chartData)
      pure mempty

    pure $ R.div { id: containerId }
```

7.7. Project Setup

Add these dependencies to your `spago.yaml`:

```
package:
  dependencies:
    - psd3-selection
    - react-basic
    - react-basic-dom
    - react-basic-hooks
```

If using local PSD3 packages:

```
workspace:
  extraPackages:
    psd3-selection:
      path: ../purescript-psd3-selection
    psd3-tree:
      path: ../purescript-psd3-tree
    psd3-graph:
```

```
      path: ../purescript-psd3-graph
```

7.8. Why Framework-Agnostic Matters

The visualization code (`barChartAST`) is completely decoupled from React. This means:

- **Reusable**: The same AST works in Halogen, React, or vanilla JS
- **Testable**: Test visualization logic without a framework
- **Portable**: Switch frameworks without rewriting visualizations

7.9. Next Steps

- React Events from PSD3 - Click handlers and state updates
- Force Simulations in React - Tick callbacks and drag
- Halogen Events from PSD3 - Compare with Halogen
- PSD3 in a Simple Web Page - No framework at all

Chapter 8. React Events from PSD3 Visualizations

When building a React application with PSD3 visualizations, you need to bridge events from the D3 world back to React's state system. This guide shows you how.

8.1. The Challenge

PSD3 visualizations live in the DOM, managed by D3. React manages your component state. When a user clicks something in the visualization, you need to:

1. Capture the event in PSD3 via behaviors
2. Extract relevant data (which element was clicked, its datum)
3. Call a React state setter
4. React re-renders to reflect the new state

8.2. Solution: The Callback Bridge

Pass React's state setter into your visualization AST:

```
import PSD3.Internal.Behavior.Types (onClickWithDatum)
import React.Basic.Hooks (useState, (/\))

mkInteractiveBarChart :: Component { chartData :: Array Number }
mkInteractiveBarChart = do
  containerId <- useContainerId

  component "InteractiveBarChart" \props -> React.do
    -- React state: which bar index is selected
    selectedBar /\ setSelectedBar <- useState (Nothing :: Maybe Int)

    useEffectOnce do
      void $ runD3 do
        let selector = "#" <> containerId
        clear selector
        container <- select selector
        -- Pass the state setter into the AST builder
        renderTree container (barChartAST props.chartData setSelectedBar)
      pure mempty

    -- React re-renders when selectedBar changes
    pure $ R.div_
      [ R.div { id: containerId }
      , R.p_ [ R.text $ case selectedBar of
            Nothing -> "Click a bar to select it"
            Just idx -> "Selected bar " <> show (idx + 1) ]
```

```
    ]
```

8.3. The AST with Click Behavior

Use `withBehaviors` to attach event handlers, and `onClickWithDatum` to access the bound data:

```
barChartAST :: Array Number -> ((Maybe Int -> Maybe Int) -> Effect Unit) -> A.AST
Number
barChartAST dataset setSelectedBar =
  A.elem SVG
    [ computed "width" (num 500.0)
    , computed "height" (num 300.0)
    ]
    `A.withChild`
      A.joinData "bars" "rect" dataset \_ ->
        A.elem Rect
          [ fromWithIndex "x" (\_ i -> toNumber i * 60.0 + 20.0)
          , fromWithIndex "y" (\val _ -> 280.0 - val * 2.5)
          , computed "width" (num 50.0)
          , fromWithIndex "height" (\val _ -> val * 2.5)
          , computedStr "cursor" (text "pointer")  -- Visual feedback
          ]
          `A.withBehaviors`
            [ onClickWithDatum \clickedValue -> do
                -- Find the index of the clicked value
                let mIdx = Array.findIndex (_ == clickedValue) dataset
                case mIdx of
                  Just idx -> setSelectedBar (\_ -> Just idx)
                  Nothing -> pure unit
            ]
```

8.4. How It Works

1. **`onClickWithDatum`** - Receives the datum bound to the clicked element (from D3's `data` property)
2. **`setSelectedBar`** - React's state setter, passed through from the component
3. **State update** - `setSelectedBar (\_ → Just idx)` triggers React re-render
4. **UI updates** - React shows the new selection in the paragraph

8.5. Available Event Behaviors

PSD3 provides several event behaviors:

```
import PSD3.Internal.Behavior.Types
  ( onClick           -- Effect Unit (no datum)
  , onClickWithDatum  -- (datum -> Effect Unit)
```

```
  , onMouseEnter      -- (datum -> Effect Unit)
  , onMouseLeave      -- (datum -> Effect Unit)
  , onMouseEnterWithInfo  -- (MouseEventInfo datum -> Effect Unit)
  , onMouseLeaveWithInfo  -- (MouseEventInfo datum -> Effect Unit)
  )
```

`MouseEventInfo` includes position data for tooltips:

```
type MouseEventInfo datum =
  { datum :: datum
  , clientX :: Number, clientY :: Number  -- Viewport-relative
  , pageX :: Number, pageY :: Number      -- Document-relative
  , offsetX :: Number, offsetY :: Number  -- Element-relative
  }
```

8.6. Pattern: Multiple Event Types

Handle different events on the same element:

```
A.elem Circle [...]
  `A.withBehaviors`
    [ onClickWithDatum \d -> setSelected (Just d.id)
    , onMouseEnter \d -> setHovered (Just d.id)
    , onMouseLeave \_ -> setHovered Nothing
    ]
```

8.7. Next Steps

- Force Simulations in React - Tick-driven updates
- Halogen Events from PSD3 - Compare with Halogen
- PSD3 in a React App - Basic integration

Chapter 9. Force Simulations in React

Force-directed layouts require continuous animation during simulation. This guide shows how to integrate PSD3's force engine with React components.

9.1. The Challenge

Force simulations run at ~60fps, updating node positions each tick. You need to:

1. Create and configure the simulation
2. Update DOM positions on each tick (fast path, not React re-renders)
3. Support drag interaction that reheats the simulation
4. Clean up properly when the component unmounts

9.2. Complete Example

```
import PSD3.ForceEngine.Simulation as Sim
import PSD3.ForceEngine.Types (ForceSpec(..))

mkForceGraph :: Component {}
mkForceGraph = do
  containerId <- useContainerId

  component "ForceGraph" \_props -> React.do
    isRunning /\ setIsRunning <- useState true

    useEffectOnce do
      -- Create sample data
      let nodes = createNodes 20
          links = createLinks nodes

      -- 1. Create simulation
      sim <- Sim.create Sim.defaultConfig

      -- 2. Set up tick callback for DOM updates
      Sim.onTick (updateDOMPositions containerId) sim

      -- 3. Configure simulation
      Sim.setNodes nodes sim
      Sim.setLinks links sim

      -- 4. Add forces
      Sim.addForce (ManyBody "charge"
        { strength: -30.0, theta: 0.9, distanceMin: 1.0, distanceMax: 1000.0 }) sim
      Sim.addForce (Link "link"
        { distance: 30.0, strength: 1.0, iterations: 1 }) sim
      Sim.addForce (Center "center"
```

```
        { x: 200.0, y: 150.0, strength: 1.0 }) sim

      -- 5. Render initial state
      void $ runD3 do
        let selector = "#" <> containerId
        clear selector
        container <- select selector
        renderTree container (forceGraphAST nodes)

      -- 6. Start simulation
      Sim.start sim

      -- 7. Attach drag behavior
      nodeElements <- Sim.querySelectorElements ("#" <> containerId <> " circle")
      Sim.attachDrag nodeElements sim

      -- 8. Cleanup on unmount
      pure do
        Sim.stop sim
        setIsRunning (\_ -> false)

    pure $ R.div
      { id: containerId
      , style: R.css { width: "400px", height: "300px" }
      }
```

9.3. The Tick Callback

The tick callback runs ~60fps during simulation. Use direct DOM manipulation for performance:

```
-- PureScript
updateDOMPositions :: String -> Effect Unit
updateDOMPositions containerId = do
  circles <- Sim.querySelectorElements ("#" <> containerId <> " circle")
  updateCirclePositions circles

-- FFI (JavaScript) - direct DOM access is fastest
foreign import updateCirclePositions :: Array Element -> Effect Unit
```

```
// JavaScript FFI
export const updateCirclePositions = (circles) => () => {
  for (const circle of circles) {
    const d = circle.__data__;  // D3's bound data
    if (d) {
      circle.setAttribute('cx', d.x);
      circle.setAttribute('cy', d.y);
    }
  }
```

```
};
```

Don't trigger React re-renders in the tick callback. Direct DOM manipulation is essential for 60fps performance.

9.4. Drag Behavior

`Sim.attachDrag` provides simulation-aware drag:

```
-- After rendering, attach drag to node elements
nodeElements <- Sim.querySelectorElements ("#" <> containerId <> " circle")
Sim.attachDrag nodeElements sim
```

This sets up D3 drag handlers that:

1. **Drag start**: Reheat simulation (alpha → 1.0)
2. **During drag**: Fix node position (`fx`/`fy` set to pointer position)
3. **Drag end**: Release node (`fx`/`fy` cleared, node floats free)

9.5. Alternative: Pinning Drag

For exploration interfaces where users want to "pin" nodes:

```
Sim.attachPinningDrag nodeElements sim
```

Behavior: - First drag: pins the node where dropped - Subsequent drags: small movement (<3px) unpins, larger movement repositions

9.6. Node Data Structure

Simulation nodes need specific fields:

```
type ForceNode =
  { id :: Int
  , x :: Number, y :: Number      -- Position (mutated by simulation)
  , vx :: Number, vy :: Number    -- Velocity (mutated by forces)
  , fx :: Nullable Number         -- Fixed x (set during drag)
  , fy :: Nullable Number         -- Fixed y (set during drag)
  , group :: Int                  -- Your custom data
  }
```

9.7. Cleanup

Always stop the simulation when the component unmounts:

```
useEffectOnce do
  -- ... setup ...

  -- Return cleanup function
  pure do
    Sim.stop sim
```

This cancels the animation frame loop and prevents memory leaks.

9.8. Next Steps

- React Events from PSD3 - Click handlers
- PSD3 in a React App - Basic integration
- Halogen Events from PSD3 - Compare approaches

Chapter 10. Simple CSS Hover Effects

For simple hover effects that only affect the hovered element, CSS is faster and simpler than JavaScript. This guide shows how to implement CSS-based hover effects with PSD3.

10.1. When to Use CSS Hover

Use CSS hover when:

- The effect only affects the hovered element itself
- You want maximum performance (no JavaScript on mouse move)
- The effect is purely visual (color, size, opacity changes)

Use JavaScript hover (see Coordinated Highlighting) when:

- Multiple elements should respond to one hover
- You need to access datum values
- Effects span multiple views

10.2. Basic CSS Hover

10.2.1. 1. Add a Class to Your Elements

```
import PSD3.Expr.Friendly (class_)

myVisualization =
  A.joinData "circles" "circle" myData \d ->
    A.elem Circle
      [ cx (from _.x)
      , cy (from _.y)
      , r (num 5.0)
      , fill (text "steelblue")
      , class_ (text "data-point")  -- Add a class for CSS targeting
      ]
```

10.2.2. 2. Define CSS Hover Rules

```
.data-point {
  transition: all 0.15s ease-out;
  cursor: pointer;
}

.data-point:hover {
  r: 8;  /* SVG attribute - grows the circle */
  fill: coral;
```

```
  filter: drop-shadow(0 2px 4px rgba(0,0,0,0.3));
}
```

10.3. Common Hover Patterns

10.3.1. Highlight on Hover

```
.node {
  transition: opacity 0.2s, stroke-width 0.2s;
}

.node:hover {
  opacity: 1 !important;
  stroke: #fbbf24;
  stroke-width: 2px;
}
```

10.3.2. Scale Up on Hover

```
.bar {
  transition: transform 0.15s ease-out;
  transform-origin: bottom center;
}

.bar:hover {
  transform: scaleY(1.05);
}
```

10.3.3. Glow Effect

```
.highlight-target {
  transition: filter 0.2s;
}

.highlight-target:hover {
  filter: drop-shadow(0 0 8px currentColor);
}
```

10.3.4. Cursor Feedback

```
.draggable {
  cursor: grab;
}
```

```
.draggable:active {
  cursor: grabbing;
}

.clickable {
  cursor: pointer;
}
```

10.4. Combining with PSD3 Behaviors

You can use CSS for visual feedback while using PSD3 behaviors for logic:

```
A.elem Circle
  [ class_ (text "data-point clickable")
  , cx (from _.x)
  , cy (from _.y)
  ]
  `A.withBehaviors`
    [ onClick \d -> handleClick d.id  -- JavaScript handles the click
    -- CSS handles the hover effect (no behavior needed)
    ]
```

```
/* CSS handles all the visual hover feedback */
.data-point {
  transition: all 0.15s;
}

.data-point:hover {
  fill: coral;
  r: 8;
}

.clickable {
  cursor: pointer;
}
```

10.5. Performance Tips

1. **Use `transform` and `opacity`** for animations - they're GPU-accelerated
2. **Avoid animating `width`, `height`, `top`, `left`** - they trigger layout
3. **Keep transitions short** (150-200ms) for responsive feel
4. **Use `will-change`** sparingly for complex animations

```
.optimized-hover {
```

```
  will-change: transform, opacity;
  transition: transform 0.15s ease-out, opacity 0.15s ease-out;
}

.optimized-hover:hover {
  transform: scale(1.1);
  opacity: 0.9;
}
```

10.6. SVG-Specific Considerations

SVG attributes like `r`, `cx`, `cy` can be animated with CSS, but browser support varies:

```
/* Works in modern browsers */
circle {
  transition: r 0.2s;
}

circle:hover {
  r: 10;
}

/* More compatible: use transform instead */
circle {
  transition: transform 0.2s;
  transform-origin: center;
}

circle:hover {
  transform: scale(1.5);
}
```

10.7. Next Steps

- Coordinated Highlighting - Multi-element hover effects
- Linked Brushing - Selection across views

Chapter 11. Coordinated Highlighting

Coordinated highlighting synchronizes visual emphasis across multiple views. When a user hovers over an element in one visualization, related elements in all views respond - some highlighted as primary, others as related, and unrelated elements dimmed.

11.1. The Two-Tier Hover System

PSD3 provides two tiers of hover behavior:

11.1.1. Tier 1: Element-Local Hover (Simple)

For basic hover effects that only affect the element itself:

```
import PSD3.Internal.Behavior.Types (onHover)

-- Simple style change on hover
T.elem Circle [...]
  `T.withBehaviors`
    [ onHover
        { enter: [{ attr: "stroke", value: "#333" }, { attr: "stroke-width", value:
"3" }]
        , leave: [{ attr: "stroke", value: "#ddd" }, { attr: "stroke-width", value:
"1.5" }]
        }
    ]
```

This is CSS-speed, element-local, and requires no coordination. Use it for simple feedback.

11.1.2. Tier 2: Coordinated Highlighting (Cross-View)

For synchronized highlighting across multiple visualizations:

```
import PSD3.Internal.Behavior.Types (HighlightClass(..), onCoordinatedHighlight)
import Data.Maybe (Maybe(..))

T.elem Circle [...]
  `T.withBehaviors`
    [ onCoordinatedHighlight
        { identify: \d -> d.name           -- What does this element represent?
        , classify: \hoveredId d ->        -- How should it respond to hover?
            if d.name == hoveredId then Primary
            else if hoveredId `elem` d.connections then Related
            else Dimmed
        , group: Nothing                   -- Coordination scope (Nothing = global)
        , tooltip: Nothing                 -- Optional tooltip (see tooltip howto)
        }
```

```
    ]
```

11.2. Core Concepts

11.2.1. Identity: What Does This Element Represent?

The `identify` function extracts a string identity from the element's datum:

```
identify: \d -> d.moduleName  -- A module node
identify: \d -> d.id          -- A generic entity
identify: \d -> show d.row <> "-" <> show d.col  -- A matrix cell
```

This identity is broadcast when the element is hovered.

11.2.2. Classification: How Should Elements Respond?

The `classify` function receives the hovered identity and the element's datum, returning a `HighlightClass`:

```
data HighlightClass
  = Primary   -- The hovered element itself (.highlight-primary)
  | Related   -- Connected/related elements (.highlight-related)
  | Dimmed    -- Unrelated elements (.highlight-dimmed)
  | Neutral   -- No change (default appearance)
```

Example classifications:

```
-- Simple same-name matching
classify: \hoveredId d ->
  if d.name == hoveredId then Primary else Dimmed

-- Relationship-aware (e.g., dependency graph)
classify: \hoveredId d ->
  if d.name == hoveredId then Primary
  else if hoveredId `elem` d.dependencies then Related
  else if d.name `elem` getReverseDeps hoveredId then Related
  else Dimmed

-- Matrix cell (row/column relationships)
classify: \hoveredId d ->
  if d.rowName == hoveredId && d.colName == hoveredId then Primary
  else if d.rowName == hoveredId || d.colName == hoveredId then Related
  else Dimmed
```

11.2.3. Groups: Scoping Coordination

By default, all elements with `group: Nothing` coordinate globally. Use named groups to isolate coordination:

```
-- These two views coordinate with each other
onCoordinatedHighlight { ..., group: Just "main-view" }

-- This view has independent highlighting
onCoordinatedHighlight { ..., group: Just "detail-panel" }
```

11.3. CSS Classes

The library applies these CSS classes based on classification:

```
/* Primary - the hovered element and its equivalents */
.highlight-primary {
  opacity: 1 !important;
  stroke: #fbbf24 !important;
  stroke-width: 2px !important;
}

/* Related - connected elements */
.highlight-related {
  opacity: 0.85 !important;
  stroke: #60a5fa !important;
  stroke-width: 1.5px !important;
}

/* Dimmed - unrelated elements */
.highlight-dimmed {
  opacity: 0.2 !important;
}
```

Customize these in your CSS to match your visualization's aesthetic.

11.4. Pattern: Enriched Datum Types

When using data joins with coordinated highlighting, create enriched datum types that carry everything needed for both rendering AND classification:

```
-- Enriched type with pre-computed relationships
type NodeDatum =
  { id :: String
  , name :: String
  , x :: Number
  , y :: Number
```

```
  , dependencies :: Array String    -- For classification
  , dependents :: Array String      -- For classification
  , cluster :: Int                  -- For coloring
  }

-- Build enriched data before the join
let enrichedNodes = nodes <#> \n ->
  { id: n.id
  , name: n.name
  , x: n.x
  , y: n.y
  , dependencies: getDeps n.id graph
  , dependents: getReverseDeps n.id graph
  , cluster: n.cluster
  }

-- Use in data join with template
T.joinData "nodes" "g" enrichedNodes $ \node ->
  T.elem Group [...]
    `T.withBehaviors`
      [ onCoordinatedHighlight
          { identify: _.name
          , classify: \hoveredId d ->
              if d.name == hoveredId then Primary
              else if hoveredId `elem` d.dependencies then Related
              else if hoveredId `elem` d.dependents then Related
              else Dimmed
          , group: Nothing
          , tooltip: Nothing
          }
      ]
```

11.5. Multiple Element Types

Different element types can participate in the same coordination with different classification logic:

```
-- Nodes: Primary if same name
nodeHighlight = onCoordinatedHighlight
  { identify: _.name
  , classify: \hoveredId d ->
      if d.name == hoveredId then Primary else Dimmed
  , group: Nothing
  , tooltip: Nothing
  }

-- Links: Related if either endpoint matches
linkHighlight = onCoordinatedHighlight
  { identify: \l -> l.source.name  -- Links identify by source
  , classify: \hoveredId l ->
```

```
      if l.source.name == hoveredId || l.target.name == hoveredId
        then Related
        else Dimmed
  , group: Nothing
  , tooltip: Nothing
  }

-- Labels: Primary if same name
labelHighlight = onCoordinatedHighlight
  { identify: _.name
  , classify: \hoveredId d ->
      if d.name == hoveredId then Primary else Dimmed
  , group: Nothing
  , tooltip: Nothing
  }
```

11.6. Important: Datum Timing

The library reads `data` fresh at event time, not at registration time. This means you can attach behaviors before D3 binds data to elements - a common pattern with the Tree API's `T.withBehaviors`.

11.7. Cleanup

When re-rendering visualizations, call `clearAllHighlights_` to reset state:

```
import PSD3.Internal.Behavior.FFI (clearAllHighlights_)

-- Before re-rendering
liftEffect clearAllHighlights_
```

This removes all highlight classes and clears the element registry.

Chapter 12. Coordinated Tooltips

Tooltips in PSD3 are integrated with the coordinated highlighting system. This enables three distinct tooltip behaviors, from traditional mouse-following tooltips to direct labels that appear on elements across views.

12.1. Quick Start

Add a tooltip to any element with coordinated highlighting:

```
import PSD3.Internal.Behavior.Types (HighlightClass(..), TooltipTrigger(..),
onCoordinatedHighlight)
import Data.Maybe (Maybe(..))

T.elem Circle [...]
  `T.withBehaviors`
    [ onCoordinatedHighlight
        { identify: \d -> d.name
        , classify: \hoveredId d ->
            if d.name == hoveredId then Primary else Dimmed
        , group: Nothing
        , tooltip: Just
            { content: \d -> d.name <> ": " <> show d.value
            , showWhen: OnHover
            }
        }
    ]
```

12.2. Tooltip Triggers

12.2.1. OnHover: Traditional Tooltip

Shows a single tooltip that follows the mouse cursor. Only the element directly under the mouse shows its tooltip.

```
tooltip: Just
  { content: \d -> d.name
  , showWhen: OnHover
  }
```

Use when: You want classic tooltip behavior - one tooltip, at the mouse position, for the element being pointed at.

Behavior:

- Tooltip appears near mouse cursor

- Moves with mouse
- Only one tooltip visible at a time
- Disappears on mouse leave

12.2.2. WhenPrimary: Direct Labels

Shows tooltips on ALL elements that classify as `Primary` across all views. This implements the "direct labeling" principle - information appears right where it's relevant.

```
tooltip: Just
  { content: \d -> d.name
  , showWhen: WhenPrimary
  }
```

Use when: You want to label corresponding elements across views. When you hover on "ModuleA" in the bubble chart, "ModuleA" labels appear on the chord diagram and matrix too.

Behavior:

- Multiple tooltips can appear simultaneously
- Each tooltip positioned near its element
- Great for showing "this is the same thing" across views
- Labels stay fixed (don't follow mouse)

12.2.3. WhenRelated: Labels on Connections

Shows tooltips on ALL elements that classify as `Primary` OR `Related`. This can show a lot of information - use judiciously.

```
tooltip: Just
  { content: \d -> d.relationship  -- e.g., "depends on" or "imported by"
  , showWhen: WhenRelated
  }
```

Use when: You want to show relationship information. Hovering on a module could label all its dependencies and dependents with their relationship type.

Behavior:

- Many tooltips can appear at once
- Can become visually busy
- Powerful for understanding connections
- Consider using for specific relationship views only

12.3. Content Functions

The `content` function receives the element's datum and returns a string:

```
-- Simple name
content: \d -> d.name

-- With value
content: \d -> d.name <> ": " <> show d.value

-- Multi-line (newlines work)
content: \d -> d.name <> "\n" <>
               "Deps: " <> show (length d.dependencies) <> "\n" <>
               "Cluster: " <> show d.cluster

-- Contextual based on element type
content: \cell ->
  if cell.value > 0.0
    then cell.rowName <> " depends on " <> cell.colName
    else cell.rowName <> " / " <> cell.colName
```

12.4. Styling Tooltips

The library creates tooltips with default styling. Override with CSS:

```
.coordinated-tooltip {
  background: rgba(15, 23, 42, 0.95);
  color: #e2e8f0;
  padding: 6px 10px;
  border-radius: 4px;
  font-size: 12px;
  font-family: system-ui, -apple-system, sans-serif;
  white-space: nowrap;
  box-shadow: 0 4px 6px -1px rgba(0, 0, 0, 0.1);
  border: 1px solid rgba(148, 163, 184, 0.2);
}

/* Make primary tooltips more prominent */
.highlight-primary + .coordinated-tooltip {
  background: rgba(251, 191, 36, 0.95);
  color: #1e293b;
}
```

12.5. Patterns and Recipes

12.5.1. Different Tooltips Per View

Each element can have its own tooltip content. Use this to show view-specific information:

```
-- Bubble pack: show metrics
bubbleTooltip = Just
  { content: \d -> d.name <> "\n" <>
                   "Size: " <> show d.size <> "\n" <>
                   "Cluster: " <> show d.cluster
  , showWhen: OnHover
  }

-- Chord diagram: show connection count
chordTooltip = Just
  { content: \d -> d.name <> "\n" <>
                   "Connections: " <> show d.connectionCount
  , showWhen: OnHover
  }

-- Matrix: show relationship
matrixCellTooltip = Just
  { content: \d -> d.rowName <> " → " <> d.colName
  , showWhen: OnHover
  }
```

12.5.2. Hover-Only Elements (No Tooltip)

Some elements might only need highlighting, not tooltips:

```
-- Links just highlight, no tooltip
onCoordinatedHighlight
  { identify: \l -> l.source.name
  , classify: \hoveredId l -> ...
  , group: Nothing
  , tooltip: Nothing  -- No tooltip for links
  }
```

12.5.3. Primary Labels + Hover Details

Combine `WhenPrimary` for labels with `OnHover` for details on different element types:

```
-- Nodes get simple labels across views
nodeConfig = onCoordinatedHighlight
  { identify: _.name
  , classify: ...
  , group: Nothing
  , tooltip: Just { content: _.name, showWhen: WhenPrimary }
  }
```

```
-- The hovered element also gets detailed tooltip
detailConfig = onCoordinatedHighlight
  { identify: _.name
  , classify: ...
  , group: Nothing
  , tooltip: Just
      { content: \d -> d.name <> "\n" <> "Full details..."
      , showWhen: OnHover
      }
  }
```

12.5.4. Conditional Tooltips

The content function can return empty strings or minimal content based on conditions:

```
tooltip: Just
  { content: \d ->
      if d.isImportant
        then d.name <> " (critical path)"
        else ""  -- No tooltip for unimportant items
  , showWhen: WhenPrimary
  }
```

12.6. Performance Considerations

- `OnHover`: Single tooltip, minimal overhead
- `WhenPrimary`: Multiple tooltips, but only on matching elements
- `WhenRelated`: Can create many tooltips - test with your data scale

For large datasets, consider:

- Using `OnHover` for most elements
- Only enabling `WhenPrimary` for key navigation elements
- Avoiding `WhenRelated` unless the relationship count is bounded

12.7. Design Philosophy

The tooltip system follows the principle of **direct labeling** from data visualization best practices:

> Labels should be placed as close as possible to the data they describe.

- `OnHover` is indirect (cursor → separate tooltip)
- `WhenPrimary` is direct (label on the element itself)

- `WhenRelated` extends direct labeling to show connections

Choose based on information density and user task:

- **Exploration**: `OnHover` keeps the view uncluttered
- **Comparison**: `WhenPrimary` shows corresponding elements
- **Understanding relationships**: `WhenRelated` reveals connections

Chapter 13. Linked Brushing

Linked brushing lets users select elements in one view and see corresponding elements highlighted in all other views. This is a powerful technique for exploring multi-dimensional data.

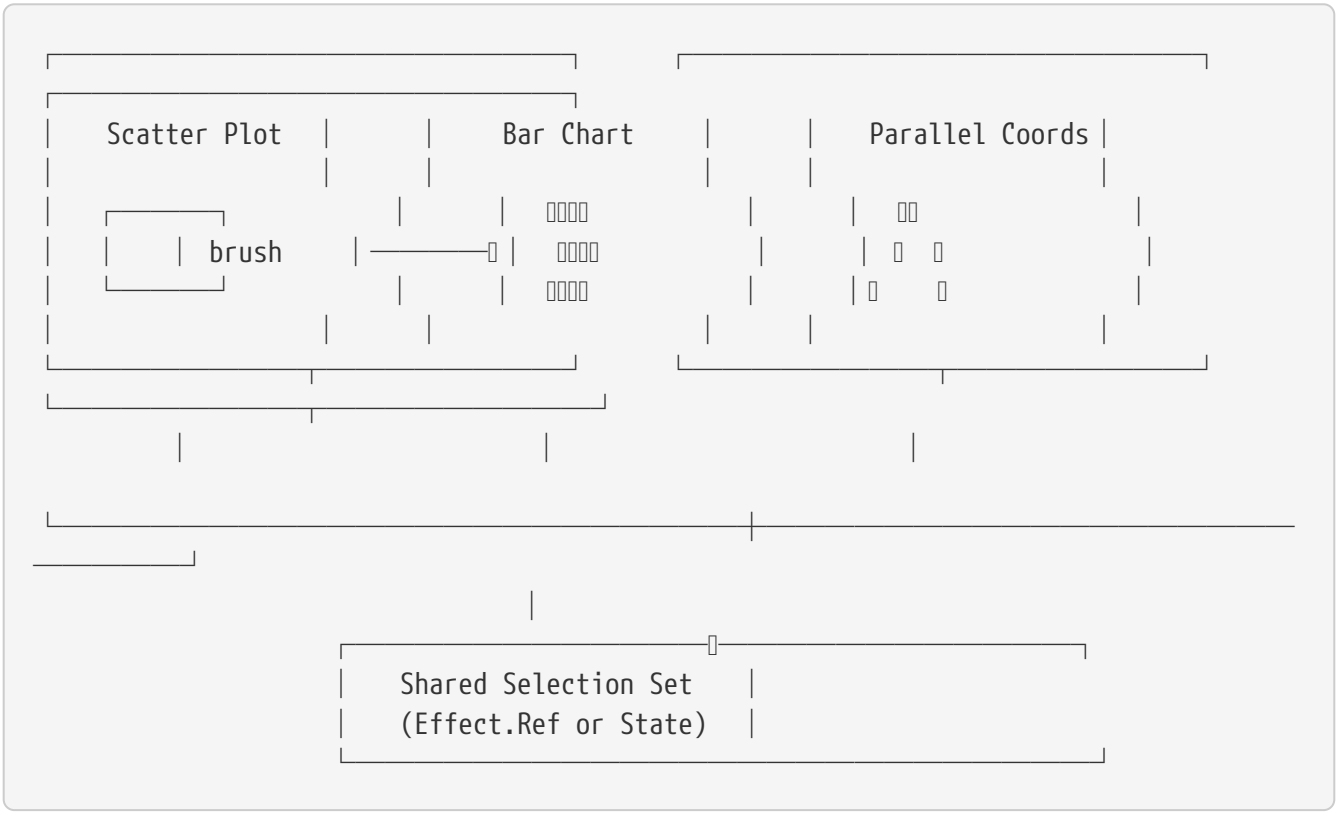
13.1. What is Linked Brushing?

When you brush (drag to select) in one view:

1. Elements in the brushed region become "selected"
2. Corresponding elements in other views are highlighted
3. Non-corresponding elements are dimmed

This creates a coordinated multi-view exploration experience.

13.2. Architecture Overview



13.3. Implementation

13.3.1. 1. Define Selection State

```
type SelectionState =
  { selectedIds :: Set String
  , brushExtent :: Maybe BrushExtent
  }
```

```
type BrushExtent =
  { x0 :: Number, y0 :: Number
  , x1 :: Number, y1 :: Number
  }
```

13.3.2. 2. Set Up the Brush Behavior

```
import PSD3.Brush (Brush, brush, brushX, brushY, onBrush, onBrushEnd)

scatterPlot :: Array DataPoint -> (Set String -> Effect Unit) -> A.AST DataPoint
scatterPlot points onSelectionChange =
  A.named SVG "scatter" [ width (num 400.0), height (num 400.0) ]
    `A.withChildren`
      [ -- Brush overlay (must be on top for events)
        A.elem Group [ class_ (text "brush") ]
          `A.withBehaviors`
            [ brush
                { extent: [[0.0, 0.0], [400.0, 400.0]]
                , onBrush: \extent -> do
                    let selected = findPointsInExtent points extent
                    onSelectionChange selected
                , onBrushEnd: \mExtent ->
                    case mExtent of
                      Nothing -> onSelectionChange Set.empty  -- Brush cleared
                      Just extent -> pure unit  -- Keep current selection
                }
            ]

      -- Data points
      , A.joinData "points" "circle" points \d ->
          A.elem Circle
            [ cx (from _.x), cy (from _.y), r (num 4.0)
            , class_ (text "data-point")
            ]
      ]

findPointsInExtent :: Array DataPoint -> BrushExtent -> Set String
findPointsInExtent points { x0, y0, x1, y1 } =
  Set.fromFoldable
    $ map _.id
    $ filter (\p -> p.x >= x0 && p.x <= x1 && p.y >= y0 && p.y <= y1) points
```

13.3.3. 3. Apply Selection to All Views

```
barChart :: Array DataPoint -> Set String -> A.AST DataPoint
barChart points selectedIds =
  A.elem SVG [ width (num 400.0), height (num 300.0) ]
```

```
    `A.withChild`
      A.joinData "bars" "rect" points \d ->
        A.elem Rect
          [ x (from \p _ -> toNumber (indexOf p points) * 20.0)
          , y (from _.value >>> negate >>> (_ + 300.0))
          , width (num 18.0)
          , height (from _.value)
          , class_ (text $ if Set.member d.id selectedIds
                           then "bar selected"
                           else "bar dimmed")
          ]
```

13.3.4. 4. Style Selected vs Dimmed

```
.data-point, .bar {
  transition: opacity 0.15s, fill 0.15s;
}

.selected {
  opacity: 1;
  fill: coral;
}

.dimmed {
  opacity: 0.2;
}

/* When nothing is selected, show everything */
.data-point:not(.selected):not(.dimmed),
.bar:not(.selected):not(.dimmed) {
  opacity: 1;
}
```

13.4. Integrating with Halogen

```
type State =
  { data :: Array DataPoint
  , selectedIds :: Set String
  }

data Action
  = SelectionChanged (Set String)

render :: State -> H.ComponentHTML Action () m
render state =
  HH.div_
    [ HH.div [ HP.ref scatterRef ] []  -- Scatter with brush
```

```
    , HH.div [ HP.ref barRef ] []      -- Bar chart
    , HH.div [ HP.ref parallelRef ] [] -- Parallel coordinates
    ]

handleAction = case _ of
  SelectionChanged ids -> do
    H.modify_ _ { selectedIds = ids }
    -- Re-render all views with new selection
    state <- H.get
    renderAllViews state
```

13.5. Brush Types

PSD3 supports several brush types:

```
-- 2D rectangular brush
brush { extent, onBrush, onBrushEnd }

-- Horizontal brush only (for bar charts, timelines)
brushX { extent, onBrush, onBrushEnd }

-- Vertical brush only
brushY { extent, onBrush, onBrushEnd }
```

13.6. Performance Considerations

For large datasets:

1. **Debounce brush events** - Don't update on every pixel
2. **Use CSS classes** instead of re-rendering for selection changes
3. **Consider WebGL** for 10k+ points

```
-- Debounced brush handler
onBrush: debounce 50 \extent -> do
  let selected = findPointsInExtent points extent
  onSelectionChange selected
```

13.7. Next Steps

- Coordinated Highlighting - Hover-based coordination
- Zoomable Visualizations - Pan and zoom

Chapter 14. Making Visualizations Zoomable

Zoom and pan allow users to explore large or detailed visualizations. This guide shows how to add zoom behavior to your PSD3 visualizations.

14.1. Basic Zoom Setup

```
import PSD3.Internal.Behavior.Types (Behavior(..), ZoomBehavior(..), ScaleExtent(..))

zoomableViz :: A.AST Unit
zoomableViz =
  A.named SVG "svg"
    [ width (num 800.0)
    , height (num 600.0)
    ]
    `A.withBehaviors`
      [ Zoom $ defaultZoom (ScaleExtent 0.5 10.0) ".zoom-group" ]
    `A.withChild`
      A.named Group "zoom-group"
        [ class_ (text "zoom-group") ]
        `A.withChildren`
          [ -- Your visualization content here
            A.elem Circle [ cx (num 400.0), cy (num 300.0), r (num 50.0) ]
          ]
```

14.2. How It Works

1. **Zoom behavior** attaches to the SVG (captures mouse/touch events)
2. **Transform** is applied to a child group (the `.zoom-group`)
3. Content inside the group moves and scales together

```
<svg>                          ← Captures zoom events
  <g class="zoom-group"        ← Receives transform
     transform="translate(100,50) scale(2)">
    <circle .../>              ← Moves with the group
    <rect .../>
  </g>
</svg>
```

14.3. Zoom Configuration

```
-- Scale extent: min and max zoom levels
ScaleExtent 0.1 10.0   -- Can zoom out to 10%, in to 1000%
ScaleExtent 1.0 5.0    -- No zoom out, up to 5x zoom in
```

```
-- Default zoom helper
defaultZoom :: ScaleExtent -> Selector -> ZoomBehavior
defaultZoom extent targetSelector = ZoomBehavior
  { scaleExtent: extent
  , target: targetSelector
  , onZoom: Nothing  -- Use default transform behavior
  }
```

14.4. Custom Zoom Handlers

For more control, provide a custom zoom handler:

```
Zoom $ ZoomBehavior
  { scaleExtent: ScaleExtent 0.5 10.0
  , target: ".zoom-group"
  , onZoom: Just \transform -> do
      -- transform :: { k :: Number, x :: Number, y :: Number }
      -- k = scale, x/y = translation

      -- Update some state based on zoom
      when (transform.k > 5.0) do
        showDetailedLabels

      -- Log zoom level
      log $ "Zoom: " <> show transform.k
  }
```

14.5. Zoom with Semantic Levels

Change what's displayed based on zoom level:

```
type ZoomLevel = Overview | Intermediate | Detailed

zoomLevelFromScale :: Number -> ZoomLevel
zoomLevelFromScale k
  | k < 1.5 = Overview
  | k < 4.0 = Intermediate
  | otherwise = Detailed

-- In your visualization, render different content per level
renderAtZoomLevel :: ZoomLevel -> Array DataPoint -> A.AST DataPoint
renderAtZoomLevel level points = case level of
  Overview ->
    -- Just show dots
    A.joinData "points" "circle" points \d ->
      A.elem Circle [ cx (from _.x), cy (from _.y), r (num 3.0) ]
```

```
  Intermediate ->
    -- Show dots with labels on hover
    A.joinData "points" "g" points \d ->
      A.elem Group []
        `A.withChildren`
          [ A.elem Circle [ cx (from _.x), cy (from _.y), r (num 5.0) ]
          , A.elem Text [ x (from _.x), y (from _.y), opacity (num 0.0) ]
              `A.withBehaviors` [ showOnHover ]
          ]

  Detailed ->
    -- Always show labels
    A.joinData "points" "g" points \d ->
      A.elem Group []
        `A.withChildren`
          [ A.elem Circle [ cx (from _.x), cy (from _.y), r (num 8.0) ]
          , A.elem Text [ x (from _.x), y (from _.y), textContent (from _.name) ]
          ]
```

14.6. Zoom to Fit

Programmatically zoom to show all content:

```
import PSD3.Internal.Behavior.FFI (zoomTo, zoomIdentity)

-- Zoom to show a specific bounding box
zoomToFit :: D3Selection -> BoundingBox -> Effect Unit
zoomToFit svg { x, y, width, height } = do
  let svgWidth = 800.0
      svgHeight = 600.0
      scale = min (svgWidth / width) (svgHeight / height) * 0.9
      translateX = svgWidth / 2.0 - scale * (x + width / 2.0)
      translateY = svgHeight / 2.0 - scale * (y + height / 2.0)
  zoomTo svg { k: scale, x: translateX, y: translateY }

-- Reset zoom to identity
resetZoom :: D3Selection -> Effect Unit
resetZoom svg = zoomIdentity svg
```

14.7. Combining Zoom with Drag

When elements are draggable inside a zoomable container:

```
-- The zoom group handles panning
A.named SVG "svg" [...]
  `A.withBehaviors` [ Zoom $ defaultZoom (ScaleExtent 0.5 5.0) ".zoom-group" ]
```

```
  `A.withChild`
    A.named Group "zoom-group" [ class_ (text "zoom-group") ]
      `A.withChild`
        -- Individual nodes are draggable
        A.joinData "nodes" "g" nodes \node ->
          A.elem Group [...]
            `A.withBehaviors`
              [ Drag SimpleDrag ]  -- Drag works in zoomed coordinates
```

14.8. Mobile Touch Support

Zoom behavior automatically handles touch events:

- **Pinch**: Two-finger pinch to zoom
- **Pan**: One-finger drag to pan (when zoom is active)
- **Double-tap**: Zoom in

No additional configuration needed.

14.9. Performance Tips

1. **Use CSS transforms** - They're GPU-accelerated
2. **Throttle zoom callbacks** for expensive operations
3. **Consider level-of-detail** to reduce rendering at low zoom

14.10. Next Steps

- Halogen Events from PSD3 - Integrate zoom with app state
- Linked Brushing - Another interaction pattern

Chapter 15. Data Joins Deep Dive

This guide is coming soon. It will cover:

- The General Update Pattern (GUP)
- `joinData` vs `updateJoin` vs `nestedJoin`
- Key functions for identity matching
- Enter/update/exit transitions
- Nested data structures

15.1. Quick Reference

```
-- Simple join (no transitions)
A.joinData "circles" "circle" myData $ \d ->
  A.elem Circle [ cx (from _.x), cy (from _.y) ]

-- With enter/update/exit
A.updateJoin "circles" "circle" myData
  (\d -> A.elem Circle [ cx (from _.x), cy (from _.y) ])
  { enter: Just { attrs: [ opacity (num 0.0) ], transition: Just fadeIn }
  , update: Just { attrs: [], transition: Just move }
  , exit: Just { attrs: [], transition: Just fadeOut }
  , keyFn: Just _.id
  }
```

15.2. Next Steps

- Building a Bar Chart
- Animated Transitions

Chapter 16. Animated Transitions

This guide is coming soon. It will cover:

- Transition configuration
- Easing functions
- Sequencing transitions
- Performance tips

16.1. Quick Reference

```
import PSD3.Internal.Transition.Types (TransitionConfig)

fadeIn :: TransitionConfig
fadeIn = { duration: 500, delay: 0, ease: "easeCubicOut" }

slideDown :: TransitionConfig
slideDown = { duration: 300, delay: 100, ease: "easeBackOut" }
```

16.2. Next Steps

- Data Joins Deep Dive

Understanding the Architecture

Chapter 17. Finally Tagless Architecture

PSD3 uses the **finally tagless** encoding for its attribute expressions. This enables the same visualization code to be interpreted in multiple ways - rendering to the DOM, generating test strings, or producing documentation.

17.1. What is Finally Tagless?

Finally tagless is a technique for building DSLs (domain-specific languages) where:

1. You define operations as type class methods
2. Different interpreters implement the type class differently
3. The same expression can be interpreted multiple ways

17.2. The Problem It Solves

Consider attribute expressions. You might want to:

- **Render** them to actual DOM attributes
- **Stringify** them for testing
- **Analyze** them for documentation

With a traditional AST (initial encoding), you'd need pattern matching for each use case. With finally tagless, each interpreter just implements the interface.

17.3. PSD3's Expression System

```
-- The type class defines what operations exist
class Expr repr where
  num :: Number -> repr Number
  text :: String -> repr String
  plus :: repr Number -> repr Number -> repr Number
  field :: forall @sym a r. IsSymbol sym => Cons sym a r r'
        => repr { | r' } -> repr a
  -- ... more operations
```

Different interpreters give different meanings:

```
-- D3 interpreter: produces actual values
instance Expr D3Expr where
  num n = D3Expr \_ _ -> n
  plus a b = D3Expr \d i -> runD3Expr a d i + runD3Expr b d i

-- String interpreter: produces descriptions
instance Expr StringExpr where
```

```
num n = StringExpr $ show n
plus a b = StringExpr $ "(" <> runStringExpr a <> " + " <> runStringExpr b <> ")"
```

17.4. Why This Matters for You

1. **Testability**: Your visualization specs can be tested without a browser
2. **Documentation**: The same code can generate human-readable descriptions
3. **Debugging**: Multiple views of what your code actually does

17.5. The Practical Upshot

You write:

```
xPosition = field @"x" `times` num 40.0 `plus` num 50.0
```

And PSD3 can:

- Evaluate it for rendering: `180.0` (given `{ x: 3.25 }`)
- Stringify it for testing: `"(x * 40.0 + 50.0)"`
- Document it: "x coordinate scaled by 40, offset by 50"

17.6. Further Reading

- The Selection AST
- Interpreters

Chapter 18. The Selection AST

PSD3 visualizations are built as **trees**. This page explains the structure and why it matters.

18.1. Trees, Not Chains

D3.js uses method chaining - each call returns a selection you continue to operate on:

```
selection.append("g").attr("class", "group").append("circle")...
```

PSD3 builds a tree data structure:

```
data Tree datum
  = Node (TreeNode datum)
  | Join { name, key, joinData, template }
  | ...
```

18.2. Node Structure

Each node contains:

```
type TreeNode datum =
  { name :: Maybe String        -- Optional ID for later retrieval
  , elemType :: ElementType     -- SVG, Circle, Rect, Group, etc.
  , attrs :: Array (Attribute datum)   -- Attributes to set
  , behaviors :: Array (Behavior datum) -- Attached behaviors
  , children :: Array (Tree datum)      -- Child nodes
  }
```

18.3. Data Joins as Tree Nodes

The `Join` variant represents a data-driven section:

```
A.joinData "circles" "circle" myData $ \d ->
  A.elem Circle [ cx (from _.x), cy (from _.y) ]
```

This creates a `Join` node with:

- `name`: "circles" (for later retrieval)
- `key`: "circle" (element type to create)
- `joinData`: the array of data
- `template`: a function from datum to subtree

18.4. Why Trees?

1. **Inspectable**: You can examine the structure before rendering
2. **Transformable**: Tree transformations are well-understood
3. **Testable**: Compare trees for equality in tests
4. **Multi-interpretable**: Render to DOM, to strings, to diagrams

18.5. Visualization

The Mermaid interpreter can show your tree structure:

```
import PSD3.Interpreter.Mermaid (toMermaid)

-- Generates a Mermaid diagram of the AST structure
diagram = toMermaid myVisualization
```

18.6. Further Reading

- Finally Tagless Architecture
- Interpreters

Chapter 19. Interpreters

The same PSD3 visualization can be interpreted multiple ways. This page covers the available interpreters and when to use each.

19.1. Available Interpreters

19.1.1. D3 Interpreter (Primary)

Renders your visualization to the DOM via D3.js.

```
import PSD3.Render (runD3, select, renderTree)

main = void $ runD3 do
  container <- select "#chart"
  renderTree container myViz
```

Use for: Production rendering in the browser.

19.1.2. English Interpreter

Produces a human-readable description of what the visualization will do.

```
import PSD3.Interpreter.English (toEnglish)

description = toEnglish myViz
-- "Create an SVG element with width 400 and height 300,
--  containing a circle at (200, 150) with radius 50..."
```

Use for: Documentation, debugging, accessibility descriptions.

19.1.3. Mermaid Interpreter

Generates a Mermaid diagram showing the AST structure.

```
import PSD3.Interpreter.Mermaid (toMermaid)

diagram = toMermaid myViz
-- "graph TD\n  A[SVG] --> B[Circle]\n  ..."
```

Use for: Visualizing complex tree structures, documentation.

19.1.4. MetaAST Interpreter

Converts the typed AST to a simpler untyped representation.

```
import PSD3.Interpreter.MetaAST (toMetaAST)

meta = toMetaAST myViz
```

Use for: Serialization, cross-language interop, tooling.

19.2. Writing Custom Interpreters

Since the AST is just data, you can write your own interpreters:

```
-- Count elements in a visualization
countElements :: forall datum. Tree datum -> Int
countElements (Node node) = 1 + sum (map countElements node.children)
countElements (Join j) = 1  -- Count the join itself
countElements _ = 1
```

19.3. The Interpreter Pattern

This is the power of the declarative approach:

1. **Describe** what you want (build the tree)
2. **Interpret** it however you need (render, test, document)

The same visualization spec serves multiple purposes without modification.

19.4. Further Reading

- Finally Tagless Architecture
- The Selection AST

Chapter 20. Reimagining the Spreadsheet with Functional Programming

A design exploration for psd3-sheetless

20.1. The Insight

Traditional spreadsheets are secretly functional programs in disguise - they just don't know it. Every formula is a pure function, the dependency graph is a dataflow network, and "fill down" is really `extend` from comonad theory.

What if we made this explicit? What if a spreadsheet was built from the ground up on:

- **Comonads** for contextual computation
- **Recursion schemes** for structured folds and unfolds
- **Lenses** for composable, bidirectional data access
- **Zippers** for navigation and focus

20.2. 1. The Spreadsheet as Comonad

A spreadsheet is a classic example of the **Store comonad**:

```
-- Store s a ≈ (s -> a, s)
-- A function from position to value, plus a current focus

class Comonad w where
  extract :: w a -> a                    -- Get the focused value
  extend :: (w a -> b) -> w a -> w b     -- Apply at every position
  duplicate :: w a -> w (w a)            -- Every cell sees the whole sheet
```

The key insight: **a formula is a function from "the spreadsheet focused at my position" to a value**.

```
-- Traditional formula
=A1 + B1

-- What it really means
\sheet -> extract (moveLeft sheet) + extract (moveLeft (moveLeft sheet))

-- Or more elegantly
\sheet -> sumOf (neighbors sheet)
```

The `extend` operation says: "apply this formula at every cell, each seeing its own local context." This is exactly what "fill down" does!

```
-- Fill a formula down a column
extend (\s -> extract (above s) * 1.1) sheet
-- Every cell becomes 1.1x the cell above it
```

20.2.1. Why This Matters

- **Formulas become first-class functions** you can compose, test, and visualize
- **Context is explicit** - a formula can access neighbors, regions, the whole sheet
- **"Fill down" has a theory** - it's comonadic extend

20.3. 2. Recursion Schemes: Folds Made Visible

Instead of hiding aggregation behind opaque functions like `SUM`, make the fold structure explicit and visualizable.

20.3.1. The Zoo of Schemes

Scheme	Type	Spreadsheet Use
cata (fold)	`F a → a`	SUM, COUNT, MAX - consume structure to value
ana (unfold)	`a → F a`	Generate sequences, date ranges, growth series
hylo	`a → b`	Generate then reduce (common pattern)
para	`F (μF, a) → a`	Fold that sees original - compare to previous
histo	`F (Cofree F a) → a`	Fold with full history - moving averages!
zygo	Two folds	Compute mean (need both sum and count)

20.3.2. Example: Histomorphism for Moving Average

A histomorphism is a fold where each step can see **all previous results**, not just the immediate one. Perfect for moving averages:

```
-- Moving average needs history of last N values
movingAvg :: Int -> Histo [] Number -> Number
movingAvg n history =
  let recent = take n (toList history)
  in sum recent / length recent

-- Apply to a column
=histo (movingAvg 3) (A1:A100)
```

The recursion scheme makes the pattern explicit and visualizable.

20.4. 3. Lenses: Composable Data Access

Traditional cell references are ad-hoc string parsing (`"A1"`, `"Sheet2!B3:C10"`). Lenses give us composable, type-safe, **bidirectional** access.

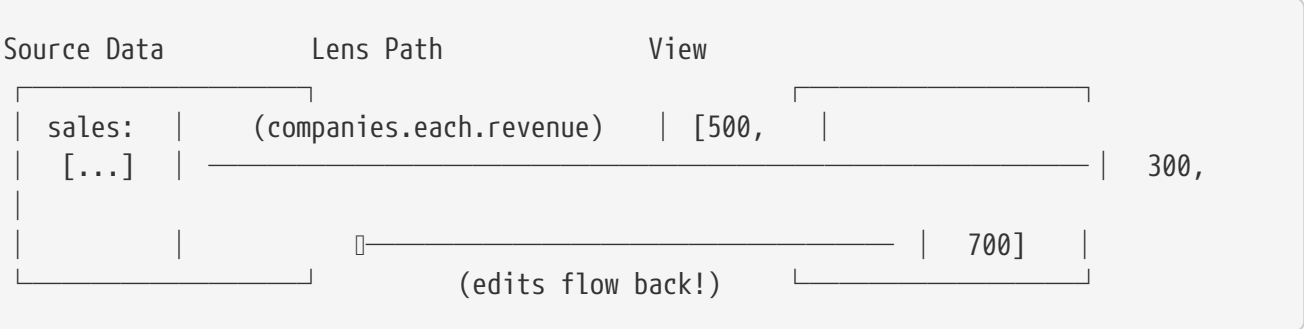
```
-- Lenses compose with (.)
revenue :: Lens' Company Number
q1 :: Lens' YearData QuarterData
amount :: Lens' QuarterData Number

-- Deep access
view (revenue . q1 . amount) companyData

-- The magic: bidirectional!
set (revenue . q1 . amount) 50000 companyData
-- Updates flow back through the lens path
```

20.4.1. Bidirectional Spreadsheets

The profound implication: if your "view" is computed through lenses, **editing the view updates the source**.



Change a number in the view, it propagates back to the source.

20.5. 4. Zippers: Focus and Navigation

A **zipper** is a data structure with a "focus" point and efficient local navigation. Selection in a spreadsheet is naturally a zipper.

```
type SheetZipper a =
  { focus :: a
  , left :: List a          -- cells to the left
  , right :: List a         -- cells to the right
  , above :: List (List a)  -- rows above
  , below :: List (List a)  -- rows below
  }

-- Navigation is O(1)
moveRight :: SheetZipper a -> Maybe (SheetZipper a)
```

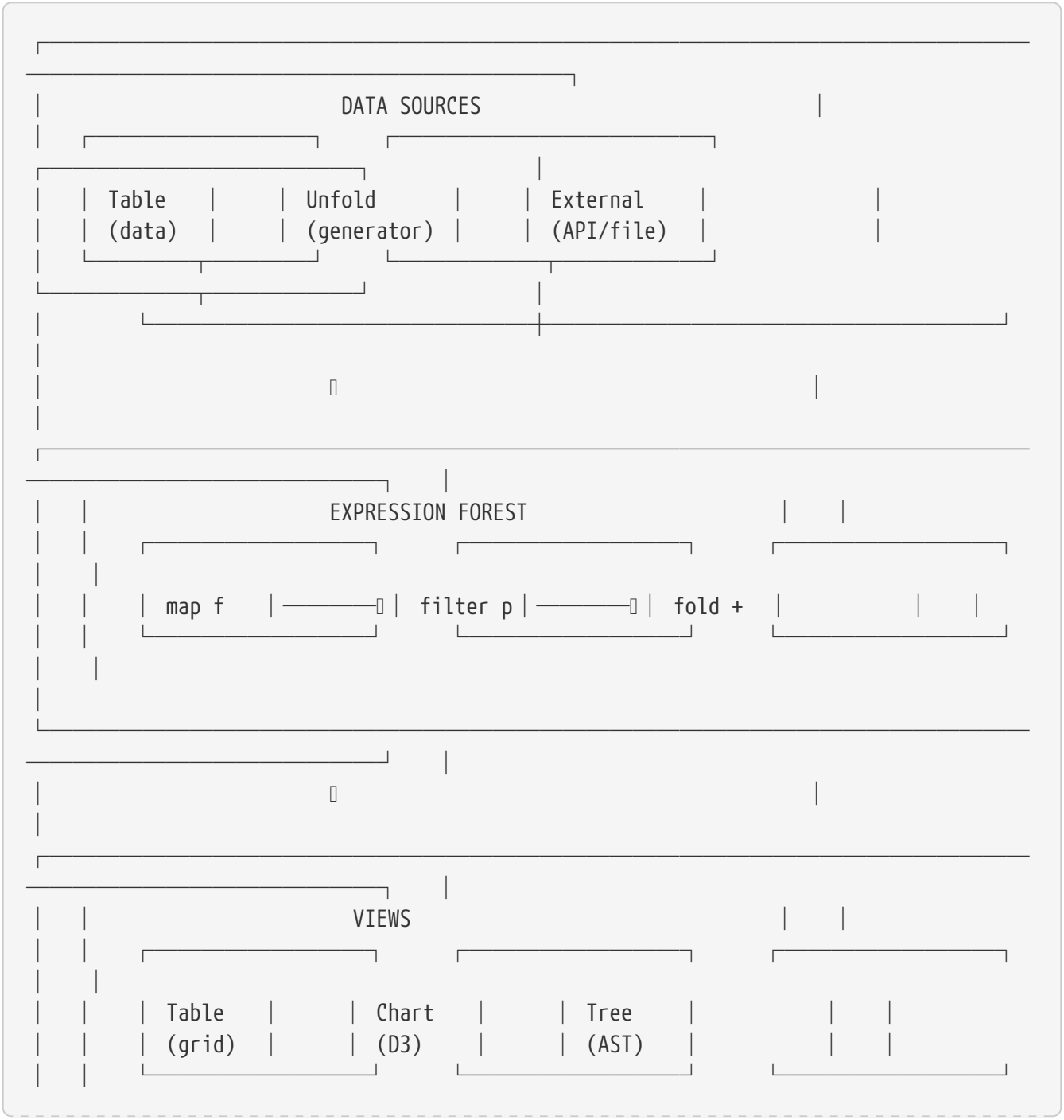
```
moveDown :: SheetZipper a -> Maybe (SheetZipper a)
```

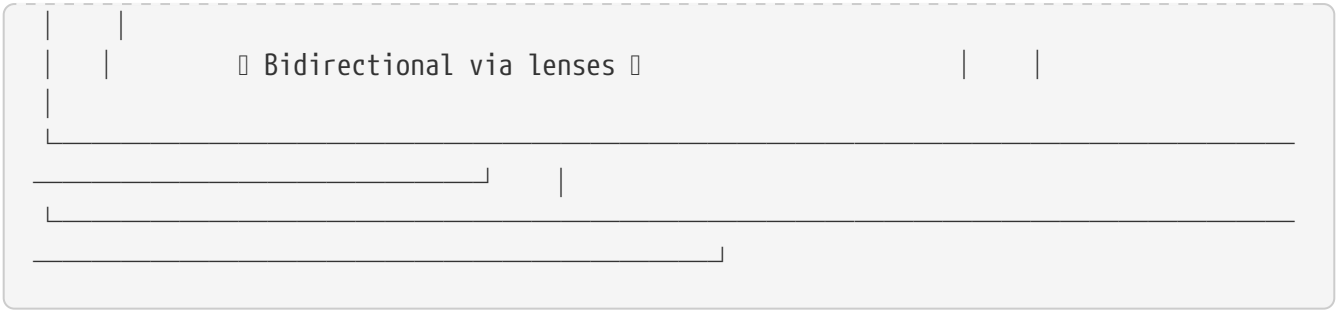
20.5.1. Connection to Comonad

A zipper is a comonad! The `duplicate` operation gives you "a zipper of zippers" - every position contains the whole structure focused at that position.

```
duplicate :: SheetZipper a -> SheetZipper (SheetZipper a)
-- Every cell now contains "the spreadsheet focused at that cell"
-- This is exactly what formulas need!
```

20.6. 5. The Architecture





20.7. References

- "Comonads for User Interfaces" - Arthur Xavier
- "Recursion Schemes" - Patrick Thomson's blog series
- "Lenses, Folds, and Traversals" - Edward Kmett
- "The Zipper" - Gérard Huet's original paper
- "Functional Pearl: The Essence of the Iterator Pattern" - Gibbons & Oliveira

20.8. See Also

- Unified Data DSL - How computation and visualization unite
- Finally Tagless Architecture - The encoding pattern

Chapter 21. Unified Data DSL: Bridging Computation and Visualization

21.1. The Core Insight

Traditional thinking separates:

- **Spreadsheets**: compute values from data
- **Visualizations**: render data to DOM

But at their core, both are doing the same thing: **transforming data through a typed pipeline**. The only difference is what comes out the other end.

```
Data Source → [map/fold/filter] → Output
                    ↓
            Spreadsheet: Values
            Visualization: DOM Trees
            Analysis: Statistics
            Export: JSON/CSV
```

This document describes a **unified finally-tagless DSL** where:

1. Data flow operations are defined once, abstractly
2. Different interpreters produce different outputs
3. The same computation can be a spreadsheet cell AND a visualization
4. Types ensure correctness across both worlds

21.2. Part 1: The Unified DataDSL

21.2.1. Core Type Class

```
-- | The fundamental operations on data, independent of output type
class DataDSL repr where
  -- Primitives
  num    :: Number -> repr Number
  str    :: String -> repr String
  bool   :: Boolean -> repr Boolean

  -- Data source binding ("join points")
  source :: forall a. DataSource a -> repr (Array a)

  -- The universal data operations
  mapA    :: forall a b. (a -> b) -> repr (Array a) -> repr (Array b)
  foldA   :: forall a b. (b -> a -> b) -> b -> repr (Array a) -> repr b
```

```
  filterA :: forall a. (a -> Boolean) -> repr (Array a) -> repr (Array a)
  zipA    :: forall a b. repr (Array a) -> repr (Array b) -> repr (Array (Tuple a b))

  -- Arithmetic
  add :: repr Number -> repr Number -> repr Number
  mul :: repr Number -> repr Number -> repr Number
```

21.2.2. Why This is Profound

This captures the essence of D3's data join:

```
// D3
selection.selectAll("rect")
  .data(myData)           // ← source
  .join("rect")
  .attr("height", d => d.value)  // ← mapA (\d -> d.value)
```

And the essence of spreadsheet formulas:

```
=MAP(A1:A10, x => x * 1.1)   -- mapA (\x -> x * 1.1) (source range)
=SUM(B1:B10)                  -- foldA (+) 0 (source range)
=FILTER(C1:C10, x => x > 0)   -- filterA (\x -> x > 0) (source range)
```

They are the same operations. Only the interpreter differs.

21.3. Part 2: Data Sources as Join Points

In D3, a data join binds data to DOM elements. In our unified model, a **join point** is where data enters a computation pipeline:

```
source :: DataSource a -> repr (Array a)

-- This single operation replaces:
-- - D3's selection.data()
-- - Spreadsheet's cell range reference
-- - Database query execution
-- - File loading
```

The key insight: **the join point is abstract**. The interpreter decides what "binding data" means:

- In a spreadsheet: look up cell values
- In D3: bind to DOM selection
- In analysis: load from database
- In testing: inject mock data

21.4. Part 3: Display as Profunctor

Display transformations are **typed morphisms** that compose:

```
-- | Percentage: Number -> String
-- | Pipeline: scale by 100 → round to 1 decimal → show → add "%"
percentageD :: forall repr. DisplayPrimitives repr => repr Number String
percentageD =
  suffixD "%" `composeD` fixedD 1 `composeD` scaleD 100.0

-- | Currency: Number -> String
currencyD :: forall repr. DisplayPrimitives repr => repr Number String
currencyD =
  prefixD "$" `composeD` fixedD 2
```

The profunctor structure gives us:

1. **Reusability**: Define `percentageD` once, use it anywhere
2. **Composability**: Chain transformations type-safely
3. **Adaptability**: `lmapD` adapts display to any source type

21.5. Part 4: The Same Computation, Multiple Interpretations

```
-- | Define a computation abstractly
quarterlyAnalysis :: forall repr. DataDSL repr =>
  DataSource Quarter -> repr (Array GrowthRate)
quarterlyAnalysis src =
  mapA computeGrowth $
    filterA (_.revenue >>> (_ > 0.0)) $
      source src

-- | Spreadsheet interpretation: produces cell values
runAsSheet :: Array GrowthRate
runAsSheet = evalSheet (quarterlyAnalysis quarterlyData)

-- | Visualization interpretation: produces a bar chart
runAsViz :: Tree GrowthRate
runAsViz = evalViz $
  join (quarterlyAnalysis quarterlyData) \rate ->
    elem Rect
      [ height (scaleY rate.value)
      , fill "steelblue"
      ]

-- | Analysis interpretation: produces statistics
runAsStats :: Statistics
```

```
runAsStats = evalStats (quarterlyAnalysis quarterlyData)
```

21.6. Part 5: Decomposing PSD3's Join Variants

21.6.1. Current PSD3 Structure (Problematic)

PSD3 currently has four join constructors:

```
data Tree datum
  = Node (TreeNode datum)
  | Join { ... }              -- Basic
  | NestedJoin { ... }        -- + decompose
  | UpdateJoin { ... }        -- + GUP behaviors
  | UpdateNestedJoin { ... }  -- + decompose + GUP
```

This is a **Cartesian product** of features, leading to exponential growth.

21.6.2. Decomposed Design (Proposed)

Factor into orthogonal, composable concerns:

```
-- | Core join: bind data to template
class JoinDSL repr where
  join :: forall a. repr (Array a) -> (a -> repr (Tree a)) -> repr (Tree a)

-- | Decomposition: extract nested data
class JoinDSL repr <= NestedJoinDSL repr where
  withDecompose :: forall outer inner.
    (outer -> Array inner) -> ...

-- | GUP behaviors: enter/update/exit
class JoinDSL repr <= GUPDSL repr where
  withGUP :: forall a. GUPBehaviors a -> ...

-- | Compose them freely
myViz =
  withGUP enterFadeIn $
    withDecompose _.points $
      join sourceData pointTemplate
```

21.7. The Philosophical Win

Data visualization and data computation are **not separate disciplines**. They are two views of the same underlying reality: **typed data flowing through transformations**.

By unifying the DSL, we:

1. Eliminate artificial boundaries between "spreadsheet people" and "visualization people"
2. Enable live links between computed values and their visual representations
3. Make the full power of functional programming available in both contexts
4. Create a foundation for tools that truly integrate analysis and presentation

21.8. See Also

- Functional Spreadsheets - The comonad foundation
- Finally Tagless Architecture - The encoding technique
- Design Decisions - Pragmatic choices

Chapter 22. Design Decisions

22.1. Analysis of puresheet

We analyzed puresheet (GPL v3) as architectural reference.

22.1.1. What Works Well

1. **AST Design** - Clean recursive ADT for expressions:
   - `FnApply`, `LambdaFn`, `InfixFnApply` - standard constructs
   - `WhereExpr`, `SwitchExpr`, `CondExpr` - bindings and control flow
   - Pattern matching with guards
2. **Type Organization** - Layered modules:
   - `SyntaxTree/` - Pure AST types
   - `Parser/` - Parsing to AST
   - `Evaluator/` - AST interpretation
   - `Printer/` - AST pretty-printing
3. **Halogen Architecture**:
   - Halogen Store for global state
   - Component-based UI
   - Clean separation of concerns

22.1.2. Coupling to Avoid

puresheet's AST is coupled to spreadsheet concepts:

```
-- From puresheet SyntaxTree/FnDef.purs
data FnBody
  = ...
  | Cell' Cell                    -- spreadsheet cell reference
  | CellValue' CellValue          -- cell literal
  | CellMatrixRange Cell Cell     -- A1:C3 ranges
```

We generalize these to data source references:

```
-- Our approach
data Expr
  = ...
  | DataRef DataSource Path       -- reference any data
  | DataRange DataSource Selection -- select from any source
```

22.2. Key Design Decisions

22.2.1. 1. Expression Language

Keep the functional core from puresheet's design:

- First-class functions, currying
- Pattern matching with guards
- Custom operators with precedence
- Local bindings (where/let)

But parameterize over the "leaf" type - what expressions can reference.

22.2.2. 2. Data Sources as First-Class

Instead of cells, expressions reference abstract data sources:

```
data DataSource
  = TableSource TableId           -- in-memory table
  | QuerySource QueryId           -- SQL query
  | WebSource WebSourceId         -- scraped web data
  | ComputedSource ExprId         -- result of another expr

data Selection
  = All                           -- entire source
  | Column ColRef                 -- single column
  | Row RowRef                    -- single row
  | Range ColRef ColRef RowRef RowRef
  | Filter Expr                   -- predicate
```

22.2.3. 3. D3-Style Data Joins

Expressions don't just **read** data - they **bind** to it:

```
data Binding
  = Enter DataSource Expr         -- new data → create
  | Update DataSource Expr        -- changed data → transform
  | Exit DataSource Expr          -- removed data → cleanup
```

This connects naturally to visualization: the expression forest **is** the visualization pipeline.

22.2.4. 4. Visualization as Output

Each expression tree has an output type:

```
data OutputType
```

```
  = DataOutput                    -- produces data (for chaining)
  | ChartOutput ChartSpec         -- produces visualization
  | TreeOutput                    -- self-visualizes as AST
```

22.3. Implementation Phases

22.3.1. Phase 1: Core Language

1. Define `Expr` ADT (generalized from puresheet's `FnBody`)
2. Define `Pattern` for pattern matching
3. Define `Literal` for primitive values
4. Parser using bookhound or similar
5. Evaluator (pure, no effects)
6. Pretty printer

22.3.2. Phase 2: Data Sources

1. `DataSource` ADT
2. `Table` - simple in-memory tables
3. Selection/filtering
4. Data change tracking (for joins)

22.3.3. Phase 3: Charting Demo

1. Integrate PSD3 D3 wrapper
2. Chart specification types
3. Expression → Chart rendering
4. Basic Halogen app

22.3.4. Phase 4: AST Visualization

1. Adapt PSD3 PatternTree
2. Expression → Tree visualization
3. Interactive exploration
4. Link to evaluation steps

22.3.5. Phase 5: Expression Forest

1. Multi-expression workspace
2. Expression dependencies
3. Data flow visualization

4. Full data join semantics

22.4. Tech Stack

- **Language**: PureScript
- **UI**: Halogen + Halogen Store
- **Parsing**: Bookhound (or purescript-parsing)
- **Visualization**: PSD3 D3 wrappers
- **Build**: Spago + esbuild

22.5. Relationship to PSD3

Reuses:

- `purescript-psd3-tree` - Tree visualization
- `purescript-psd3-selection` - Selection model
- `purescript-psd3-layout` - Layout algorithms
- D3 FFI bindings from psd3-demo-website

New:

- Expression language and evaluator
- Data source abstraction
- Expression forest workspace

22.6. See Also

- Unified Data DSL - The unifying vision
- Finally Tagless Architecture - The encoding

Chapter 23. Graph Algorithms Library Design

A companion library to `psd3-simulation` for graph algorithms, analysis, and layout.

23.1. Core Types

```
-- Graph representations
newtype NodeId = NodeId String
type Edge r = { from :: NodeId, to :: NodeId | r }
type WeightedEdge = Edge (weight :: Number)

-- Multiple representations for different use cases
data Graph a =
  | AdjacencyList (Map NodeId (Array { neighbor :: NodeId | a }))
  | EdgeList (Array (Edge a))
```

23.2. Pathfinding Algorithms

Algorithm	Use Case	Complexity
A*	Single-source with heuristic	O((V+E) log V)
Dijkstra	Single-source, no heuristic	O((V+E) log V)
Bellman-Ford	Handles negative weights	O(VE)
Floyd-Warshall	All-pairs shortest paths	$O(V^3)$
BFS	Unweighted shortest path	O(V+E)

```
-- With tracing for visualization
findPathWithTrace :: NodeId -> NodeId -> Graph ->
  { result :: PathResult
  , explored :: Array NodeId      -- nodes visited
  , frontier :: Array NodeId      -- final frontier
  , steps :: Array SearchStep     -- for animation!
  }
```

Visualization potential: Animate the search process - show explored nodes, current frontier, backtracking.

23.3. Connectivity Analysis

```
-- Connected components
connectedComponents :: Graph -> Array (Set NodeId)
```

```
-- For directed graphs
stronglyConnectedComponents :: Graph -> Array (Set NodeId)  -- Tarjan's

-- Critical infrastructure
bridges :: Graph -> Array Edge           -- edges whose removal disconnects
articulationPoints :: Graph -> Array NodeId  -- nodes whose removal disconnects
```

Visualization potential: Color components differently, highlight bridges/articulation points.

23.4. Centrality Measures

These are **gold** for visualization - size/color nodes by importance.

```
type Centrality = Map NodeId Number

degreeCentrality :: Graph -> Centrality          -- simple: count edges
betweennessCentrality :: Graph -> Centrality     -- how often on shortest paths
closenessCentrality :: Graph -> Centrality       -- average distance to all others
eigenvectorCentrality :: Graph -> Centrality     -- connected to important nodes
pageRank :: Number -> Graph -> Centrality        -- the classic
```

Visualization potential:

- Node size = centrality
- Color gradient by centrality
- "Most important nodes" highlighting

23.5. Community Detection

```
-- Louvain algorithm (fast, good results)
detectCommunities :: Graph -> Array (Set NodeId)

-- With modularity score
detectCommunitiesWithScore :: Graph ->
  { communities :: Array (Set NodeId), modularity :: Number }

-- Hierarchical community structure
type Dendrogram = Tree (Set NodeId)
hierarchicalCommunities :: Graph -> Dendrogram
```

Visualization potential: Color by community, nested circle packing, collapsible clusters.

23.6. Graph Metrics

```
type GraphMetrics =
  { nodes :: Int
  , edges :: Int
  , density :: Number              -- edges / max possible edges
  , averageDegree :: Number
  , diameter :: Maybe Int          -- longest shortest path
  , radius :: Maybe Int            -- min eccentricity
  , averagePathLength :: Number
  , clusteringCoefficient :: Number
  }

analyze :: Graph -> GraphMetrics
```

23.7. Layout Algorithms

```
type Layout = Map NodeId { x :: Number, y :: Number }

-- Hierarchical (for DAGs, trees)
sugiyamaLayout :: Graph -> Layout

-- Simple deterministic layouts
circularLayout :: Graph -> Layout
gridLayout :: Graph -> Layout

-- Tree-specific
radialTreeLayout :: NodeId -> Graph -> Layout
tidyTreeLayout :: NodeId -> Graph -> Layout
```

23.8. API Design: Tracing is Optional

```
-- Standard API (most users)
module Data.Graph.Pathfinding where

findPath :: NodeId -> NodeId -> Graph -> PathResult
dijkstra :: NodeId -> Graph -> Map NodeId Number
bfs :: NodeId -> Graph -> Array NodeId

-- Traced API (for visualization, opt-in import)
module Data.Graph.Pathfinding.Traced where

findPathTraced :: NodeId -> NodeId -> Graph -> TracedResult PathResult
```

Users who just want algorithms import `Data.Graph.Pathfinding`. Users building visualizations also

import `Data.Graph.Pathfinding.Traced`.

23.9. Integration with PSD3

```
-- Example: size by PageRank
centrality <- pageRank 0.85 graph
let sizeFromRank node = 5.0 + 20.0 * (fromMaybe 0.0 $ Map.lookup node.id centrality)
setNodeSizes sizeFromRank simulation
```

23.10. Phased Implementation

23.10.1. Phase 1: Consolidation

- Create psd3-graph repo
- Move `Data.Graph.Algorithms` from psd3-selection
- Move tree utilities from psd3-tree
- Add A* from psd3-astar-demo

23.10.2. Phase 2: Pathfinding Suite

- Dijkstra (A* without heuristic)
- BFS/DFS with parent tracking
- Traced variants of all algorithms

23.10.3. Phase 3: Analysis

- Centrality measures (degree, betweenness)
- Cycle detection
- Graph metrics
- Connected components

23.10.4. Phase 4: Advanced

- Community detection (Louvain)
- PageRank
- Strongly connected components (Tarjan's)

23.10.5. Phase 5: Layouts

- Sugiyama (hierarchical)
- Radial tree
- Circular

23.11. Demo Ideas

1. **Algorithm Visualizer**: Step through A*/Dijkstra/BFS, show exploration
2. **Social Network Analyzer**: Load graph, compute centralities, detect communities
3. **Dependency Graph**: Topological sort, cycle detection, critical path
4. **Network Resilience**: Remove nodes/edges, show component fragmentation

23.12. See Also

- Zoomable Visualizations - For large graphs
- The Selection AST - Tree structures

Chapter 24. Future Directions

Captured for later exploration - the "show people things they didn't know they could do" tier

24.1. Pedagogic Recursion Schemes

The comonadic spreadsheet abstraction makes recursion schemes approachable:

- Fold trees that visualize how SUM actually computes
- Histomorphisms for moving averages (each step sees its history)
- The "Context View" showing what a cell can see (neighbors, position)
- `extend` as the theory behind "fill down"

This has real teaching potential - spreadsheets as a gateway to FP concepts.

24.2. Spreadsheets as Build Systems

Reference: "Build Systems à la Carte" by Mokhov, Mitchell, Peyton Jones (2018)

Key insight: Excel and Make are both build systems with different rebuild strategies:

- Excel: minimal rebuild based on dependency graph
- Make: topological ordering with timestamps

Our functional spreadsheet could model:

- **Dependency tracking** via the expression graph
- **Incremental computation** via selective re-evaluation
- **Build policies** as different evaluation strategies

This extends naturally to:

- CI/CD pipeline visualization
- Infrastructure-as-spreadsheet (the Kubernetes joke… but maybe not a joke?)
- Data pipeline orchestration

24.3. Cellular Automata Demo

Game of Life is already implemented in Data.Sheet and Data.Zipper:

```
life :: Sheet Boolean -> Boolean
life s = let alive = extract s
             count = length (filter identity (neighbors s))
         in if alive then count == 2 || count == 3 else count == 3
```

```
-- One generation
extend life sheet
```

Visual demo: animate the grid, show the comonadic context for each cell.

24.4. The Full Vision

```
Entry Point (familiar)     →    Intermediate           →    Advanced
_____________________________________________________________________________
_____________________________________________________

HOFs in formulas               Fold visualization          Comonadic extend
=MAP(range, fn)                See SUM's tree              Neighbor contexts
=FILTER(range, pred)           Recursion scheme picker     Build system modeling
SQL results as tables          Lens-based data paths       Cellular automata
PSD3 visualizations            Bidirectional editing       Infrastructure DSL
```

> First, do what they already do, but better. Then show them what else is possible.

24.5. See Also

- Functional Spreadsheets - The comonad foundation
- Unified Data DSL - The unifying architecture

Reference

Chapter 25. Module Overview

This page provides an overview of the module structure. For complete API documentation, see Pursuit.

25.1. Core Modules

`PSD3`

Re-exports the essential API. Start here for most use cases.

`PSD3.AST`

Tree building: `elem`, `named`, `withChild`, `joinData`, etc.

`PSD3.Expr.Friendly`

Attribute DSL: `num`, `text`, `width`, `cx`, `fill`, etc.

`PSD3.Render`

DOM rendering: `runD3`, `select`, `renderTree`.

25.2. Interpreters

`PSD3.Interpreter.D3`

The primary DOM renderer.

`PSD3.Interpreter.English`

Human-readable descriptions.

`PSD3.Interpreter.Mermaid`

AST structure diagrams.

`PSD3.Interpreter.MetaAST`

Untyped AST conversion.

25.3. Internal Modules

These are implementation details but useful for advanced use:

`PSD3.Internal.Behavior.Types`

Behaviors: `Drag`, `Zoom`, `onCoordinatedHighlight`.

`PSD3.Internal.Selection.*`

Low-level selection operations.

`PSD3.Internal.Transition.*`

Animation primitives.

25.4. Data Modules

`PSD3.Data.Node`

`SimulationNode` for force layouts.

`PSD3.Data.Tree`

Tree data structures.

`PSD3.Data.Graph`

Graph structures and algorithms.

25.5. Expression System

`PSD3.Expr.Expr`

Core expression types.

`PSD3.Expr.Attr`

Attribute constructors.

`PSD3.Expr.Interpreter.*`

Expression interpreters (Eval, SVG, CodeGen).

25.6. Scale Module

`PSD3.Scale`

D3 scale wrappers: `linear`, `ordinal`, `band`, color schemes.

25.7. Import Patterns

25.7.1. Basic Visualization

```
import PSD3.AST as A
import PSD3.AST (ElementType(..))
import PSD3.Expr.Friendly (num, text, width, height, cx, cy, r, fill)
import PSD3.Render (runD3, select, renderTree)
```

25.7.2. With Behaviors

```
import PSD3.Internal.Behavior.Types
  (Behavior(..), defaultDrag, defaultZoom, onCoordinatedHighlight)
```

25.7.3. With Scales

```
import PSD3.Scale (linear, domain, range, applyScale)
```