PSD3 v3 Exploration: Finally Tagless Attributes

The Problem

Current PSD3 architecture uses finally tagless for Tree structure but ADT for attributes:

```
-- Tree level: Finally tagless ✓
-- Multiple interpreters: D3, Mermaid, English, SemiQuine

-- Attribute level: ADT ✗
data Attribute datum
  = StaticAttr AttributeName AttributeValue
  | DataAttr AttributeName (datum -> AttributeValue)  -- Opaque function!
  | IndexedAttr AttributeName (datum -> Int -> AttributeValue)
```

When `DataAttr` holds a function, we can evaluate it but can't inspect it. This is why SemiQuine TreeToCode can only show evaluated values (`cx 0.0`) rather than the original expression (`cx (scaleX d.x)`).

This is the **Expression Problem** - we hit a boundary where the finally tagless approach stops.

The Solution: Tagless All The Way Down

If attributes were also finally tagless, we'd have full introspection:

```
class AttrDSL repr where
  cx :: repr Number -> repr Attr
  cy :: repr Number -> repr Attr
  radius :: repr Number -> repr Attr
  fill :: repr String -> repr Attr
  -- ...

class NumExpr repr where
  lit :: Number -> repr Number
  field :: String -> repr Number      -- Access datum field
  scale :: Scale -> repr Number -> repr Number
  add :: repr Number -> repr Number -> repr Number
  mul :: repr Number -> repr Number -> repr Number
  -- ...

class StringExpr repr where
  str :: String -> repr String
  fieldStr :: String -> repr String   -- Access string field
  concat :: repr String -> repr String -> repr String
  -- ...
```

Usage would change from:

```
-- Current (v2)
cx (scaleX d.x)

-- Proposed (v3)
cx (scale scaleX (field "x"))
```

Interpreter Implementations

D3 Interpreter (evaluates to DOM operations)

```
newtype D3Eval datum a = D3Eval (datum -> a)

instance numExprD3 :: NumExpr (D3Eval datum) where
  lit n = D3Eval (\_ -> n)
  field name = D3Eval (\datum -> unsafeGet name datum)
  scale s expr = D3Eval (\datum -> runScale s (runD3Eval expr datum))
  add a b = D3Eval (\d -> runD3Eval a d + runD3Eval b d)
```

CodeGen Interpreter (generates PureScript code)

```
newtype CodeGen a = CodeGen String

instance numExprCodeGen :: NumExpr CodeGen where
  lit n = CodeGen (show n)
  field name = CodeGen ("d." <> name)
  scale s expr = CodeGen (scaleName s <> " (" <> runCodeGen expr <> ")")
  add a b = CodeGen ("(" <> runCodeGen a <> " + " <> runCodeGen b <> ")")
```

English Interpreter (generates descriptions)

```
newtype English a = English String

instance numExprEnglish :: NumExpr English where
  lit n = English (show n)
  field name = English ("the " <> name <> " field")
  scale s expr = English ("scaled " <> runEnglish expr <> " using " <>
scaleName s)
```

Benefits

1. True Round-Tripping

SemiQuine could generate actual compilable code that reproduces the visualization, not just evaluated snapshots.

2. Units Support

```
class NumExpr repr where
  px :: Number -> repr Pixels
  em :: Number -> repr Em
  percent :: Number -> repr Percent
  vh :: Number -> repr ViewportHeight

  -- Type-safe operations
  addPx :: repr Pixels -> repr Pixels -> repr Pixels

  -- Conversions
  emToPx :: repr Em -> Number -> repr Pixels  -- base font size
```

The D3 interpreter would resolve units to final pixel values. The CodeGen interpreter would preserve unit strings (`"10px"`, `"2em"`). A validation interpreter could check unit consistency.

3. Optimization Passes

An optimizer interpreter could simplify expressions:

- `add (lit 0) x` → `x`
- `mul (lit 1) x` → `x`
- Constant folding: `add (lit 2) (lit 3)` → `lit 5`

4. Static Analysis

- Dependency tracking: which fields does this attribute use?
- Validation: does this expression make sense for this element type?

Challenges

1. Typed Field Access

`field "x"` is stringly-typed. Options:

- Accept runtime field lookup (current approach essentially)
- Type-level field names: `field @"x" :: repr (Field datum "x")`
- Row polymorphism for record access

2. Ergonomics

Writing `scale scaleX (field "x")` is more verbose than `scaleX d.x`. Could potentially use Template PureScript or macros if available.

3. Migration

Would require rewriting attribute expressions throughout codebase. Could potentially have compatibility shim that converts v2 style to v3.

4. Boilerplate

More typeclass instances to write and maintain. Though PureScript's deriving capabilities could help.

Tradeoffs Summary

Aspect	Current (v2)	Proposed (v3)
Syntax	`scaleX d.x`	`scale scaleX (field "x")`
Round-trip code gen	Values only	Full expressions
Units	Not supported	Type-safe units
Optimization	Not possible	Interpreter-based
Implementation	Simple	More complex
D3 familiarity	Closer to D3	More abstract

Recommendation

This is a significant architectural change worth exploring for a future major version. The benefits (true code generation, units, optimization) are substantial, but the ergonomic cost needs careful consideration.

Could prototype in a separate branch to evaluate real-world usability before committing.

The Hard Problem: Typed Field Access

This is the critical challenge that could make or break the ergonomics.

The Problem

We want `field "x"` to:

1. Know that datum has an `x` field (compile-time safety)
2. Know the type of `x` (Number, String, etc.)
3. Generate correct code (`d.x`)
4. Evaluate correctly at runtime

But `field :: String -> repr Number` has no connection to the datum type.

Option 1: Stringly Typed (Easy, Unsafe)

```
field :: String -> repr Number
field "x"  -- No compile-time check that datum has x
```

Pros: Simple, familiar Cons: Runtime errors, no autocomplete, loses PSD3's type safety promise

Option 2: Symbol + Row Polymorphism (The PureScript Way)

```
field :: forall datum r a sym
       . IsSymbol sym
      => Row.Cons sym a r datum
      => Proxy sym
      -> repr a

-- Usage:
field (Proxy :: Proxy "x")

-- Or with visible type applications (if available):
field @"x"
```

Pros: Full type safety, compiler verifies field exists and has correct type Cons: Verbose syntax, Proxy boilerplate

Option 3: Record Syntax with Custom Accessor

```
-- Define a "liftable" record accessor
class FieldAccess datum repr where
  (.) :: repr datum -> (forall r. { x :: a | r } -> a) -> repr a

-- Usage becomes:
d . _.x

-- Where d :: repr datum, and _.x is the record accessor
```

This piggybacks on PureScript's record accessor syntax.

Pros: Looks almost like `d.x` Cons: Requires `d` to be in scope as `repr datum`, not actual datum

Option 4: Typed Datum DSL

```
-- Datum is also part of the DSL
class DatumDSL repr datum where
  self :: repr datum

class HasField datum (field :: Symbol) a | datum field -> a where
  getField :: Proxy field -> repr datum -> repr a

-- Usage:
getField @"x" self
```

Option 5: Code Generation / TH-style

Pre-generate field accessors for known datum types:

```
-- Generated for type ParabolaPoint = { x :: Number, y :: Number }
parabolaX :: forall repr. NumExpr repr => repr ParabolaPoint -> repr
Number
parabolaY :: forall repr. NumExpr repr => repr ParabolaPoint -> repr
Number
```

Pros: Perfect type safety, nice syntax Cons: Requires code generation step, less flexible

Recommended Approach for Prototype

Start with **Option 2** (Symbol + Row) for type safety, but wrap it:

```
-- Internal: fully typed
fieldSym :: forall datum r a sym repr
          . IsSymbol sym
         => Row.Cons sym a r datum
         => NumExpr repr
         => Proxy sym
         -> repr datum
         -> repr a

-- User-facing: convenient wrapper for common case
-- In a module specific to their datum type:
x :: forall repr. NumExpr repr => repr ParabolaPoint -> repr Number
x = fieldSym (Proxy :: Proxy "x")

y :: forall repr. NumExpr repr => repr ParabolaPoint -> repr Number
y = fieldSym (Proxy :: Proxy "y")

-- Usage becomes:
cx (scale scaleX (x self))
```

Users define short accessors for their datum fields once, then use them naturally.

Getting Started: Minimal Prototype

Phase 1: Core DSL (1-2 hours)

Start with just numbers and one attribute:

```
-- src/PSD3v3/Expr.purs
module PSD3v3.Expr where

class NumExpr repr where
  lit :: Number -> repr Number
  add :: repr Number -> repr Number -> repr Number
```

```
  mul :: repr Number -> repr Number -> repr Number

class AttrExpr repr where
  cx :: repr Number -> repr Attr
```

Phase 2: Two Interpreters (1-2 hours)

```
-- D3 Interpreter
newtype Eval a = Eval a

runEval :: forall a. Eval a -> a
runEval (Eval a) = a

instance numExprEval :: NumExpr Eval where
  lit n = Eval n
  add (Eval a) (Eval b) = Eval (a + b)
  mul (Eval a) (Eval b) = Eval (a * b)

-- CodeGen Interpreter
newtype CodeGen a = CodeGen String

runCodeGen :: forall a. CodeGen a -> String
runCodeGen (CodeGen s) = s

instance numExprCodeGen :: NumExpr CodeGen where
  lit n = CodeGen (show n)
  add (CodeGen a) (CodeGen b) = CodeGen ("(" <> a <> " + " <> b <> ")")
  mul (CodeGen a) (CodeGen b) = CodeGen ("(" <> a <> " * " <> b <> ")")
```

Phase 3: Datum Access (The Hard Part)

```
-- Add datum threading
class NumExpr repr => DatumExpr repr datum where
  field :: forall a sym r
         . IsSymbol sym
        => Row.Cons sym a r datum
        => Proxy sym
        -> repr a

-- D3 needs datum in scope
newtype EvalD datum a = EvalD (datum -> a)

-- CodeGen just generates string
instance datumExprCodeGen :: DatumExpr CodeGen datum where
  field proxy = CodeGen ("d." <> reflectSymbol proxy)
```

Phase 4: Integration Test

Write the parabola example in v3 style and verify:

1. D3 interpreter produces working visualization
2. CodeGen interpreter produces compilable PureScript

Success Criteria

If you can write:

```
myAttr :: forall repr. (NumExpr repr, DatumExpr repr Point) => repr Attr
myAttr = cx (mul (lit 2.0) (field @"x"))
```

And get both:

- Working D3 rendering
- Generated code: `cx (2.0 * d.x)`

Then the approach is viable.

Related Work

- Oleg Kiselyov's finally tagless papers
- Phil Freeman's PureScript finally tagless examples
- Typed Template Haskell for code generation
- F# type providers for typed field access
- PureScript Record library for row polymorphism