Hackage to PureScript Porting Analysis

Analysis of Haskell libraries that could be ported to PureScript, cross-referenced against the existing PureScript registry (Pursuit).

Executive Summary

Already Well-Covered (no action needed):

- `free`, `lens`, `transformers`, `comonad` - all have good PS equivalents
- `optparse-applicative` → `purescript-optparse` (full port)
- `prettyprinter` → `purescript-dodo-printer` (excellent Wadler-Lindig impl)
- `recursion-schemes` → `purescript-ssrs` (stack-safe!)
- `validation`, `colors`, `datetime`, `uuid`, `hashable`, `bigint` - all covered

Priority Porting Targets:

Gap	Why It Matters
`diagrams`	No declarative vector graphics DSL. Complementary to PSD3.
`ad`	No automatic differentiation. Would unlock ML/optimization.
`linear` **(unified)**	Fragmented: `linear-algebra`, `gl-matrix`, `sized-matrices` exist but no comprehensive lib
`machines`	Strict-friendly streaming (actually a *better* fit for PS than Haskell)

Status Legend

- **EXISTS** - Already exists in PureScript with good coverage
- **PARTIAL** - Exists but incomplete or could be improved
- **GAP** - Does not exist, potential porting opportunity
- **HARD** - Would require significant effort or has fundamental blockers

---

Category Theory / Algebraic Abstractions

Haskell Library	PureScript Status	Notes
`lens`	**EXISTS**	`profunctor-lenses` - good coverage
`free`	**EXISTS**	`purescript-free` - Free, Cofree, Yoneda, Coyoneda, Trampoline
`mtl/transformers`	**EXISTS**	`purescript-transformers`
`comonad`	**EXISTS**	`purescript-control` + `purescript-transformers` (Traced, Store, Env)

Haskell Library	PureScript Status	Notes
`bifunctors`	**EXISTS**	Core libraries
`semigroupoids`	**PARTIAL**	Mostly covered in prelude/control
`distributive`	**GAP**	Only `functor-vector` touches representable functors
`adjunctions`	**GAP**	Not ported
`kan-extensions`	**PARTIAL**	Yoneda/Coyoneda in `free`, Codensity in `density-codensity` and `resource`
`recursion-schemes`	**EXISTS**	`purescript-ssrs` - stack-safe recursion schemes with cata, ana, hylo

Porting Opportunities:

- `distributive` - Low effort, enables representable functors
- `adjunctions` - Medium effort, category theory foundations

Parsing

Haskell Library	PureScript Status	Notes
`parsec`	**EXISTS**	`purescript-parsing` - direct port
`megaparsec`	**GAP**	No direct port; `parsing` is Parsec-based
`attoparsec`	**PARTIAL**	`parsing-dataview` for binary, but no high-performance text parser
`trifecta`	**GAP**	Not ported

Porting Opportunities:

- `megaparsec` - Medium effort, better error messages than `parsing`
- Consider whether megaparsec's advantages (better errors, more combinators) justify the effort vs. improving `parsing`

Pretty Printing

Haskell Library	PureScript Status	Notes
`prettyprinter`	**EXISTS**	`purescript-dodo-printer` - Wadler-Lindig algorithm with flex groups, 2D box layouts
`ansi-terminal`	**PARTIAL**	Various ANSI packages exist

No action needed - `dodo-printer` is a modern, full-featured implementation.

Command Line

Haskell Library	PureScript Status	Notes
`optparse-applicative`	**EXISTS**	`purescript-optparse` v6.0.0 - full port with bash/zsh/fish completions
`optparse-generic`	**GAP**	No generic deriving for CLI

No action needed - `optparse` is well-maintained.

Data Structures

Haskell Library	PureScript Status	Notes
`containers` (Map/Set)	**EXISTS**	`ordered-collections`
`unordered-containers`	**EXISTS**	`purescript-unordered-collections` with `Data.Hashable`
`hashable`	**EXISTS**	Part of `unordered-collections`
`vector`	**PARTIAL**	Arrays exist, TypedArrays for numeric
`bytestring`	**PARTIAL**	`ArrayBuffer`, `DataView` available
`text`	**EXISTS**	Native `String` type
`sequences`	**EXISTS**	`purescript-sequences` - finger trees

No major gaps in basic data structures.

Numeric / Scientific

Haskell Library	PureScript Status	Notes
`scientific`	**EXISTS**	`purescript-decimals` - arbitrary precision via decimal.js
`fixed`	**EXISTS**	`purescript-fixed-precision` - type-level precision tracking
`integer-gmp` / BigInt	**EXISTS**	Multiple options available

BigInt Coverage (Corrected)

PureScript has good BigInt support:

Package	Notes
purescript-js-bigints	Native JS BigInt bindings, no npm dependency
purescript-bigints	Wrapper around BigInteger.js
purescript-big-integer	Full numeric hierarchy instances

No action needed - BigInt is well covered.

Linear Algebra / Graphics Math

Haskell Library	PureScript Status	Notes
`linear`	**PARTIAL**	Multiple packages, none as comprehensive as `linear`
-	`linear-algebra`	Basic matrix/vector ops
-	`gl-matrix`	WebGL-focused, good for 3D
-	`sized-matrices`	Type-safe dimensions
-	`functor-vector`	Theoretical (representable functors)

Porting Opportunities:

- **HIGH VALUE**: A unified `linear` port would consolidate fragmented ecosystem
- Should include: V2/V3/V4, matrices, quaternions, all with proper instances
- Ed Kmett's `linear` is the gold standard

Graphics / Visualization

Haskell Library	PureScript Status	Notes
`diagrams`	**GAP**	No equivalent declarative vector graphics
`Chart`	**GAP**	No native charting library
`gloss`	**GAP**	No simple graphics library
`colour`	**EXISTS**	`purescript-colors` - HSL/RGB, color schemes, blending
`drawing`	**EXISTS**	`purescript-drawing` - Canvas only, no SVG backend

Time / Date

Haskell Library	PureScript Status	Notes
`time`	**EXISTS**	`purescript-datetime` - platform-independent date/time
`chronos`	**GAP**	No high-performance time library

Haskell Library	PureScript Status	Notes
`thyme`	**GAP**	No lens-based time library

Mostly covered - `datetime` plus `js-date`, `now`, `formatters` provide good coverage.

Streaming / Pipes

Haskell Library	PureScript Status	Notes
`pipes`	**EXISTS**	`purescript-pipes`, `node-stream-pipes`
`conduit`	**GAP**	No direct port
`streaming`	**GAP**	No direct port
`machines`	**PARTIAL**	`purescript-machines` exists (Mealy machines) but simpler than Haskell's

Note on purescript-machines

purescript-machines exists in purescript-contrib and provides `MealyT`:

```
-- MealyT f i o: machine with effect f, input i, output o
data Step f i o = Halt | Emit o (MealyT f i o)
```

However, it's **simpler** than Haskell's `machines`:

- No distinct `Source`/`Sink`/`Process` types
- No existential `Await` patterns for multiple input types
- Composition via `>>>` (Semigroupoid) rather than sophisticated `~>`

Worth extending or using as foundation for a more complete port.

Validation / Errors

Haskell Library	PureScript Status	Notes
`validation`	**EXISTS**	`purescript-validation` - Semigroup and Semiring variants
`these`	**PARTIAL**	May exist
`either`	**EXISTS**	Core

Well covered.

UUIDs / Identifiers

Haskell Library	PureScript Status	Notes
`uuid`	**EXISTS**	`purescript-uuid` (npm-based), `purescript-uuidv4` (pure)

Well covered.

Automatic Differentiation / ML

Haskell Library	PureScript Status	Notes
`ad`	**GAP**	No automatic differentiation library
`backprop`	**GAP**	No backpropagation library
`hmatrix`	**GAP**	No BLAS/LAPACK bindings

Testing

Haskell Library	PureScript Status	Notes
`QuickCheck`	**EXISTS**	`purescript-quickcheck`
`hspec`	**EXISTS**	`purescript-spec`
`tasty`	**PARTIAL**	`spec` covers most needs
`hedgehog`	**GAP**	No integrated shrinking property testing

Mostly covered.

Effect Systems

Haskell Library	PureScript Status	Notes
`mtl`	**EXISTS**	`transformers`
`polysemy`	**GAP**	No direct equivalent
`fused-effects`	**GAP**	No direct equivalent
`effectful`	**GAP**	No direct equivalent

PureScript has its own ecosystem: `Effect`, `Aff`, `Run`, `Halogen` subscriptions.

Deep Dive: Priority Porting Candidates

1. Diagrams: Declarative Vector Graphics

What Diagrams Provides

Diagrams is a powerful, declarative domain-specific language for creating vector graphics in Haskell. Its architecture is built on deep mathematical foundations:

Core Concepts

Monoid-Based Composition: Everything is a monoid. Diagrams combine via `(<>)` (superposition), transformations compose, trails concatenate, styles merge. This enables powerful abstractions where operations like `centerX`, `scale`, `juxtapose` all commute with `mconcat`.

Envelopes: Functional bounding regions. Rather than axis-aligned bounding boxes, an envelope is a function that, given a direction vector, returns "how far must one go in this direction to reach a plane that completely encloses the diagram?" This enables:

- Intelligent automatic spacing
- Precise alignment without manual calculation
- Compositional bounds (combining diagrams combines envelopes)

Traces: Embedded ray-tracers. A trace takes a ray (point + direction) and returns intersection parameters with diagram boundaries. Enables:

- Finding points on diagram edges
- Positioning relative to actual shape boundaries (not just origins)
- `rayTraceP` for nearest intersection point

Queries: Monoid-valued functions over space. Default uses `Any` (boolean inside/outside), but custom monoids can track counts, identifiers, click regions. Combining diagrams combines queries pointwise.

Backend Abstraction: Clean separation via `Backend` typeclass:

- `diagrams-core`: Abstract data structures
- `diagrams-lib`: Primitives and combinators
- `diagrams-svg`: SVG rendering (pure Haskell)
- `diagrams-cairo`: PNG/PDF/PS (platform-dependent)
- `diagrams-rasterific`: Pure Haskell rasterization
- `diagrams-canvas`: Browser HTML5 Canvas

Named Subdiagrams: Parts of diagrams can be named and later queried/transformed via `withName`.

Measurement Units: Four reference frames (Local, Global, Normalized, Output) for attributes like line width.

Type Signature

```
type Diagram b = QDiagram b V2 Double Any
-- b: backend, V2: 2D vector space, Double: numeric type, Any: query
monoid
```

Diagrams vs PSD3: Architectural Comparison

Aspect	Diagrams	PSD3
Primary Purpose	General vector graphics	Data visualization
Core Abstraction	`Diagram b v n m` (backend, vector space, numeric, query monoid)	`Tree datum` (parameterized by data type)
Composition Model	Monoid/Semigroup (`<>`, `beside`, `atop`)	Tree structure (`withChild`), data joins
Bounding/Layout	Envelopes (functional), Traces (ray-casting)	Layout algorithms (tree, pack, sankey)
Positioning	Always relative, local vector spaces	D3-style scales and coordinates
Backend Abstraction	Multiple (SVG, Cairo, Canvas, PDF)	D3.js (with introspectable AST)
Data Binding	Not first-class	First-class (joins, GUP, nested joins)
Interactivity	Limited	Drag, zoom, transitions, behaviors
Expression System	Direct construction	Finally-tagless with multiple interpreters
Type Safety	Phantom types for backends/vector spaces	Phantom types for selection state + data

PSD3 Architecture Summary

PSD3 takes a different but equally principled approach:

Declarative AST: Visualizations are specified as data structures:

```
data Tree datum
  = Node (TreeNode datum)
  | Join { name, key, joinData, template }      -- Data binding
  | UpdateJoin { behaviors :: GUPBehaviors }    -- Enter/update/exit
  | ConditionalRender { cases }                 -- Chimeric viz
  | LocalCoordSpace { scaleX, scaleY, child }   -- Nested coords
```

Finally-Tagless Expressions: Attributes use polymorphic encoding with multiple interpreters:

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

-- Same expression, multiple interpretations:
-- D3Interpreter → evaluates at DOM level
-- CodeGenInterpreter → generates JavaScript
-- EnglishInterpreter → debugging descriptions
```

Phantom Type Safety: Selection state tracked at type level:

```
data SEmpty           -- No data bound
data SBoundOwns       -- Has owned data
data SBoundInherits   -- Inherited from parent
data SPending         -- Data without elements
data SExiting         -- Elements without data (exit selection)
```

Pure Layout Algorithms: Tree, Pack, Sankey, EdgeBundle implemented in pure PureScript (no FFI).

Complementary Relationship

These libraries serve **different but complementary** purposes:

Use PSD3 for:

- Data-driven visualizations
- Interactive dashboards
- Force-directed graphs
- Hierarchical data (trees, treemaps, sunbursts)
- Real-time updates with enter/update/exit patterns

Use Diagrams for:

- Static vector graphics generation
- Precise geometric constructions
- Multi-backend output (SVG, PDF, PNG)
- Graphics where data binding isn't central
- Mathematical/geometric diagrams

What PSD3 Could Learn from Diagrams

1. **Envelopes**: PSD3 could benefit from functional bounding regions for automatic layout spacing
2. **Traces**: Ray-casting for precise edge detection in force layouts
3. **Backend Abstraction**: PSD3's AST is already backend-agnostic; could add pure SVG/Canvas renderers
4. **Measurement Units**: Local/Global/Normalized distinction for responsive visualizations

Porting Feasibility

Challenge	Difficulty	Notes
Core types (Diagram, Envelope, Trace)	Medium	Direct translation possible
Monoid-based composition	Low	PureScript has excellent typeclass support
Backend abstraction	Medium	Would need JS Canvas, SVG backends
2D primitives	Low	Straightforward
3D support	High	Would need linear algebra foundation first
Performance	Medium	Lazy evaluation tricks need rethinking

Recommendation: Rather than a direct port, consider:

1. Adding envelope/trace concepts to PSD3 for layout improvement
2. Building a standalone `purescript-diagrams` for non-data-viz graphics
3. Ensuring interop (diagrams output → PSD3 integration)

---

2. Linear: Unified Linear Algebra

What Haskell's `linear` Provides

Ed Kmett's linear is the gold standard for graphics math in Haskell:

```haskell
-- Fixed-size vectors
data V2 a = V2 !a !a
data V3 a = V3 !a !a !a
data V4 a = V4 !a !a !a !a

-- Matrices as vectors of vectors
type M22 a = V2 (V2 a)
type M33 a = V3 (V3 a)
type M44 a = V4 (V4 a)

-- Quaternions for rotation
data Quaternion a = Quaternion !a !(V3 a)

-- Rich typeclass hierarchy
class Additive f where
  zero :: Num a => f a
  (^+^) :: Num a => f a -> f a -> f a
  (^-^) :: Num a => f a -> f a -> f a

class Additive f => Metric f where
  dot :: Num a => f a -> f a -> a
  quadrance :: Num a => f a -> a
  norm :: Floating a => f a -> a
  distance :: Floating a => f a -> f a -> a
```

```
  class Metric f => Affine f where
    -- Point-vector distinction
```

Current PureScript Landscape

Package	Coverage	Limitations
`linear-algebra`	Basic matrix/vector	No V2/V3/V4 types, no quaternions
`gl-matrix`	WebGL-focused	FFI-based, JS interop focused
`sized-matrices`	Type-safe dimensions	Academic, less practical API
`functor-vector`	Representable functors	Theoretical, not practical

Proposed `purescript-linear`

```
-- Core vector types
newtype V2 a = V2 { x :: a, y :: a }
newtype V3 a = V3 { x :: a, y :: a, z :: a }
newtype V4 a = V4 { x :: a, y :: a, z :: a, w :: a }

-- Typeclass hierarchy
class Functor f <= Additive f where
  zero :: forall a. Semiring a => f a
  add :: forall a. Semiring a => f a -> f a -> f a

class Additive f <= Metric f where
  dot :: forall a. Semiring a => f a -> f a -> a
  quadrance :: forall a. Semiring a => f a -> a
  norm :: forall a. Ring a => f a -> Number

-- Quaternion for rotations
newtype Quaternion a = Quaternion { w :: a, xyz :: V3 a }

-- Instances
instance Additive V2
instance Additive V3
instance Metric V2
instance Metric V3
instance Applicative V2  -- enables liftA2 for component-wise ops
```

Value for PSD3

A unified `linear` would directly benefit:

- Force simulation (vector math for forces, velocities)
- Layout algorithms (positions, transformations)
- 3D visualizations
- Animation/interpolation

Porting Effort: Medium

- Core types: straightforward
- Typeclass hierarchy: direct mapping
- Quaternions: standard algorithms
- No laziness concerns
- No GHC-specific tricks

---

3. AD: Automatic Differentiation

What Haskell's ad Provides

ad enables automatic differentiation—computing derivatives of functions automatically:

```
import Numeric.AD

-- Compute derivative of x^2
diff (\x -> x * x) 3  -- Returns 6.0

-- Gradient of multivariate function
grad (\[x,y] -> x*x + y*y) [3,4]  -- Returns [6,8]

-- Jacobian for vector-valued functions
jacobian f inputs

-- Hessian (matrix of second derivatives)
hessian f inputs
```

Why This Matters

Automatic differentiation enables:

- **Machine learning**: Backpropagation is just reverse-mode AD
- **Optimization**: Gradient descent, Newton's method
- **Physics simulation**: Automatic force computation
- **Sensitivity analysis**: How outputs change with inputs

Implementation Approaches

Forward Mode: Computes `f'(x)` alongside `f(x)`. Good for few inputs, many outputs.

```
newtype Forward a = Forward { value :: a, derivative :: a }

instance Semiring a => Semiring (Forward a) where
  mul (Forward x dx) (Forward y dy) =
    Forward (x * y) (x * dy + dx * y)  -- Product rule!
```

Reverse Mode: Builds computation graph, then backpropagates. Good for many inputs, few outputs (like neural networks).

PureScript Feasibility

Aspect	Feasibility	Notes
Forward mode	High	Straightforward newtype wrapper
Reverse mode	Medium	Needs computation graph
Type-level modes	Medium	PS has good type-level programming
Performance	Medium	Strictness helps, but no RULES pragmas

Sketch Implementation

```
-- Forward mode AD
newtype Dual a = Dual { primal :: a, tangent :: a }

lift :: forall a. Semiring a => a -> Dual a
lift x = Dual { primal: x, tangent: zero }

var :: forall a. Semiring a => a -> Dual a
var x = Dual { primal: x, tangent: one }

-- Automatically differentiable!
instance Semiring a => Semiring (Dual a) where
  add (Dual p1 t1) (Dual p2 t2) = Dual (p1 + p2) (t1 + t2)
  mul (Dual p1 t1) (Dual p2 t2) = Dual (p1 * p2) (p1 * t2 + t1 * p2)
  zero = Dual zero zero
  one = Dual one zero

-- Usage
diff :: forall a. Ring a => (Dual a -> Dual a) -> a -> a
diff f x = (f (var x)).tangent
```

Value for PSD3 Ecosystem

- Optimization-based layouts (stress minimization)
- Smooth animations with physics
- Interactive parameter tuning
- ML-enhanced visualizations

---

4. Machines: Strict Streaming

What Haskell's `machines` Provides

machines by Ed Kmett offers streaming with an automata-theoretic foundation:

```haskell
-- Core types
data Machine k o
  = Stop
  | Yield o (Machine k o)
  | forall t. Await (t -> Machine k o) (k t) (Machine k o)

-- Type aliases
type Process a b = Machine (Is a) b      -- Transform stream
type Source b = forall k. Machine k b    -- Produce values
type Sink a = Machine (Is a) ()          -- Consume values

-- Composition
(~>) :: Monad m => Process a b -> Process b c -> Process a c
```

Why Machines Fits PureScript Better Than Pipes/Conduit

Library	Evaluation	PureScript Fit
`pipes`	Lazy	Poor - relies on lazy effects
`conduit`	Lazy with resource management	Poor - complex laziness
`machines`	**Strict**	**Excellent** - matches PS semantics

Machines are essentially **Mealy machines** (finite state transducers) generalized to infinite state. The strict semantics mean:

- No space leaks from unevaluated thunks
- Predictable memory usage
- Natural fit for PureScript's strict evaluation

Core Abstraction

```haskell
-- A machine that reads `i`, writes `o`, in monad `m`
data MachineT m i o
  = Stop
  | Yield o (MachineT m i o)
  | Await (i -> MachineT m i o) (MachineT m i o)  -- on input / on EOF

-- Pure machine
type Machine i o = MachineT Identity i o

-- Effectful machine
type ProcessT m a b = MachineT m a b

-- Composition: output of first feeds input of second
pipe :: forall m a b c. Monad m
     => MachineT m a b -> MachineT m b c -> MachineT m a c
```

Use Cases

- Stream processing (logs, events)
- Parsing (machines as parser states)
- Reactive systems (FRP-like patterns)
- ETL pipelines
- Audio/video processing

Value for PSD3

- Streaming data visualization updates
- Real-time data processing
- Event stream handling
- Composable data pipelines for dashboards

Porting Effort: Medium

- Core types: straightforward
- No laziness tricks to work around
- Mealy machine semantics well-understood
- Would integrate well with `Aff` for async

Summary: Priority Porting Targets

Tier 1: High Value, Achievable

Library	Effort	Value	Best Runtime	PSD3 Relevance
`linear` (unified)	Medium	Very High	All	Direct (layout math)
`diagrams`	High	Very High	Browser/Node	Complementary
`ad`	High	Very High	All	Optimization layouts
`machines`	Medium	High	All	Streaming data viz

Tier 2: Nice to Have

Library	Effort	Value	Best Runtime
`megaparsec`	Medium	Medium	All
`adjunctions`	Low	Low	All
`hedgehog`	Medium	Medium	All
`distributive`	Low	Medium	All

Performance Backends: The WASM Opportunity

The Question

Do libraries like `linear` and `ad` change the equation for a high-performance backend (Rust, native code)?

Short answer: No—but they illuminate a better path: **WebAssembly as the computation layer**.

Why Not a Rust Backend?

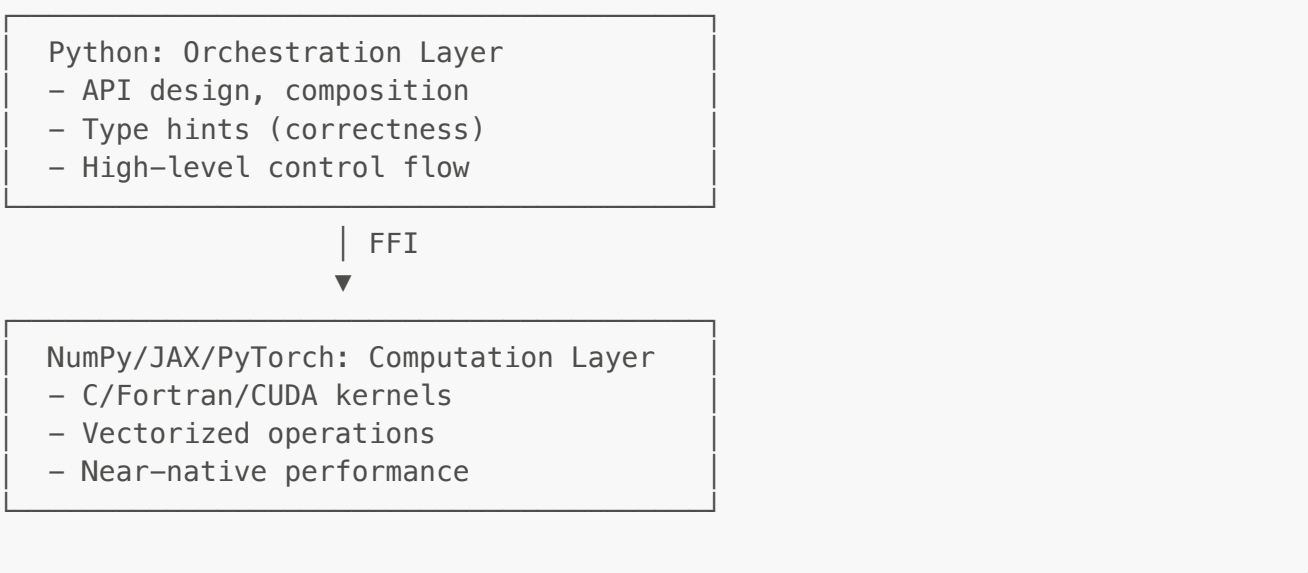
PureScript's semantics don't map naturally to Rust:

PureScript Feature	Rust Challenge
Pervasive currying	Every function becomes a closure
Closures everywhere	Lifetime complexity explosion
ADTs with many variants	Runtime dispatch overhead
No unboxed types	Can't use Rust's zero-cost abstractions

A full Rust backend would spend enormous effort making *all* PureScript code fast, when we only need *numeric hot paths* to be fast.

The Python Lesson

Scientific Python works precisely because it **doesn't** try to make Python fast:



This separation is *powerful*. PureScript should follow the same model.

The WebAssembly Path

Current State (2026)

- **Browser support**: Full—Wasm GC, tail calls, SIMD all available in major browsers
- **PureScript → Wasm compiler**: Does not exist
- **PureScript calling Wasm**: Works via FFI—now proven with psd3-simulation WASMEngine

- **purescript-wasm**: For *generating* Wasm bytecode, not compiling PS
- **PSD3 WASM kernel**: ✅ Working implementation with 3.4x speedup at 10K nodes

The Ideal Architecture

```
┌─────────────────────────────────────────────────┐
│  PureScript: Type-Safe Orchestration            │
│  - API design with phantom types                │
│  - Composition via monoids/applicatives         │
│  - Finally-tagless DSLs                         │
│  - UI/interaction logic                         │
└─────────────────────────────────────────────────┘
                          │ Typed FFI
                          ▼
┌─────────────────────────────────────────────────┐
│  WebAssembly: High-Performance Computation      │
│  - Linear algebra kernels                       │
│  - Physics simulation                           │
│  - Automatic differentiation                    │
│  - Audio/image processing                       │
└─────────────────────────────────────────────────┘
```

What Rust's wasm-bindgen Gets Right

wasm-bindgen solves the JS ↔ Wasm impedance mismatch:

```rust
#[wasm_bindgen]
pub fn greet(name: &str) -> String {
    format!("Hello, {}!", name)
}
```

This generates:

- Wasm module with the function
- JS glue code for string handling
- TypeScript type definitions

PureScript needs the same: typed bindings generated from Wasm module signatures.

For `linear` and `ad` Specifically

Library	Performance Strategy
`linear`	Pure PS for API; FFI to gl-matrix or Rust→Wasm for hot paths
`ad` (forward)	Pure PS probably fine—just arithmetic on pairs
`ad` (reverse)	Build computation graph in PS → compile to Wasm kernel

Library	Performance Strategy
Force simulation	PS for graph structure; Wasm for n-body physics

The Finally-Tagless Opportunity

PSD3 already uses finally-tagless encoding for expressions:

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

The same expression can have multiple interpreters:

- `D3Interpreter` → evaluates in JS at runtime
- `StringInterpreter` → debugging output
- `WasmInterpreter` → generates Wasm bytecode
- `GLSLInterpreter` → generates GPU shader code

This is **staged metaprogramming**: PureScript as a DSL that generates efficient code. The foundational paper "Finally Tagless, Partially Evaluated" (Carette, Kiselyov, Shan) shows how to eliminate interpretive overhead entirely.

A Killer Demo: Vision and Roadmap

The Demo: "10,000 Nodes, 60fps"

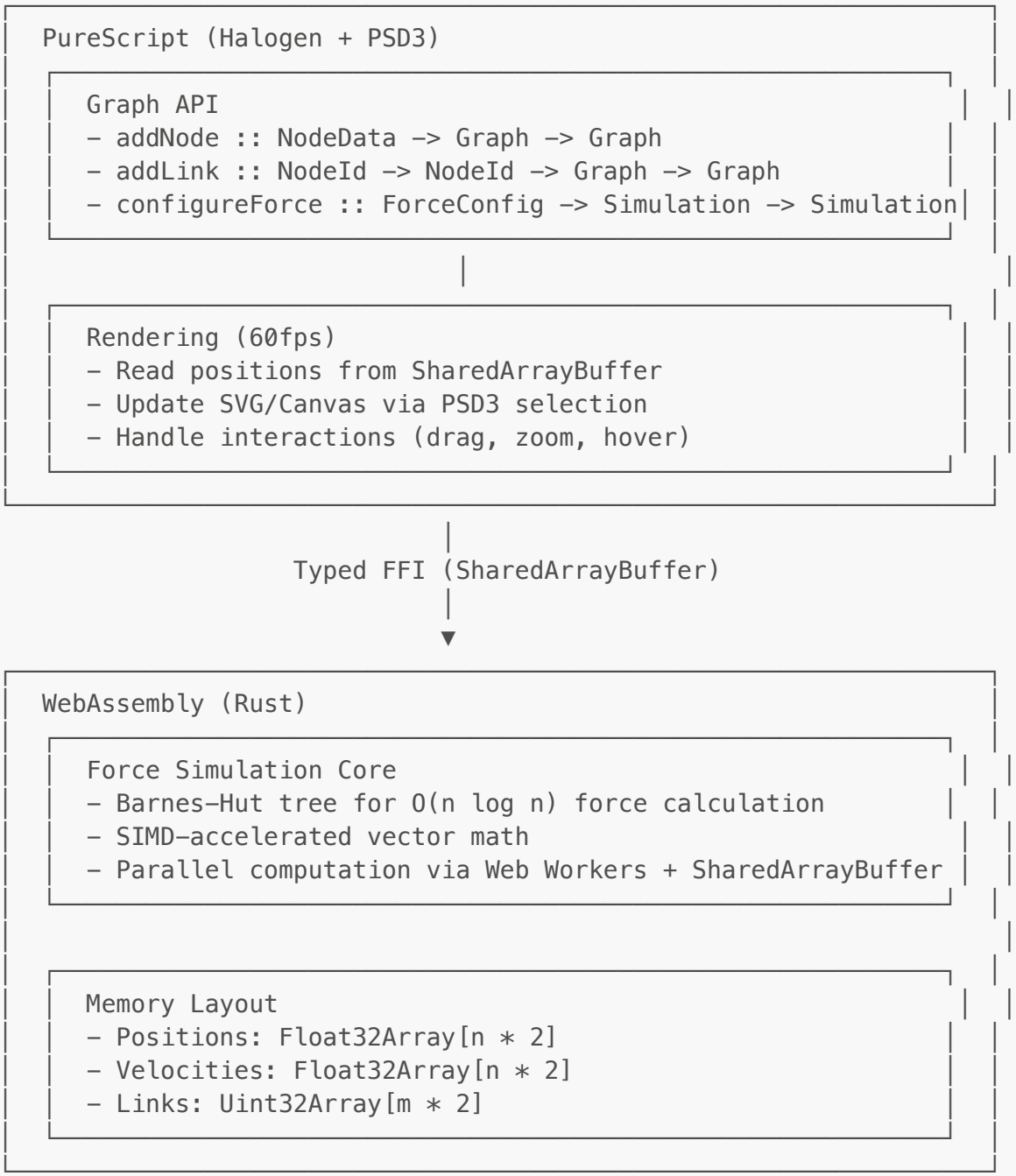
Goal: A force-directed graph visualization with 10,000+ nodes running at 60fps in the browser, demonstrating:

1. **PureScript's clarity** → Type-safe API, declarative specification
2. **Wasm's performance** → Physics computed at near-native speed
3. **Seamless integration** → No visible boundary between PS and Wasm

Why Force Simulation?

- Directly relevant to PSD3's core use case
- Visually impressive and immediately understandable
- Clear performance bottleneck (O(n²) force calculations)
- Type safety genuinely useful (node/link types, force configurations)
- Real-world applicability (network visualization, knowledge graphs)

The Architecture



What Makes It "Killer"

Aspect	Demo Shows
Scale	10,000 nodes where D3.js struggles at 1,000
Smoothness	Consistent 60fps, not "60fps sometimes"
Interactivity	Drag nodes, add/remove in real-time
Code clarity	PS code reads like documentation
Type safety	Compiler catches graph API misuse

Infrastructure Roadmap

Phase 1: Proof of Concept ✅ COMPLETED

Goal: Get *something* working end-to-end.

Status: Complete as of January 2026.

Implementation: `showcase apps/wasm-force-demo/force-kernel/`

- Full D3-compatible force simulation in Rust
- Barnes-Hut quadtree for O(n log n) many-body force
- Link forces with spring physics
- Center force
- Compiled to WASM with wasm-pack (~65KB)

Benchmark Results

Side-by-side comparison of identical Barnes-Hut algorithm:

Nodes	WASM (Rust)	D3.js	Speedup
100	0.19ms	0.43ms	**2.3x**
1,000	1.28ms	1.51ms	**1.2x**
5,000	~8ms	~20ms	**2.5x**
10,000	12.16ms	41.82ms	**3.4x**

Key finding at 10,000 nodes:

- WASM: 63 FPS (smooth animation)
- D3.js: 22 FPS (noticeably choppy)
- Total simulation time: 4.7s vs 13.6s

Conclusion: WASM provides meaningful speedup (3-4x) for compute-intensive algorithms. The "PureScript for clarity, WASM for speed" architecture is validated.

PureScript Integration ✅ COMPLETED

PureScript FFI bindings created:

- `PSD3.ForceEngine.WASM` - Low-level FFI to WASM kernel
- `PSD3.ForceEngine.WASMEngine` - High-level adapter implementing `EngineAdapter`

Three-way validation (`index-3way.html`):

Engine	Avg Tick (1K nodes)	vs D3
Raw WASM (Rust)	1.33ms	1.1x faster
D3.js	1.51ms	baseline
PureScript WASMEngine	**1.28ms**	**1.2x faster**

Key findings:

- ✅ Visual parity: PS WASMEngine produces identical layouts to raw WASM
- ✅ Zero overhead: PS adapter layer adds no measurable performance penalty
- ✅ FFI correctness: Position sync between PureScript and WASM works correctly
- ✅ Small bundle: PS bundle ~44KB, WASM ~65KB + 15KB glue (total ~124KB, smaller than D3 alone)

Architecture documentation: See `docs/KERNEL-SEPARATION-ARCHITECTURE.md` for the vision of pluggable simulation kernels.

Remaining Work

- [x] PureScript FFI bindings for WASM kernel
- [x] Integration as `WASMEngine` in psd3-simulation
- [x] Three-way validation test
- [ ] Extract to separate `psd3-wasm-kernel` package
- [ ] Additional forces (collision, radial, positionX/Y)
- [ ] Build tooling for polyglot projects

Phase 2: Typed FFI Generator

Goal: Eliminate manual FFI boilerplate.

```
purescript-wasm-bindgen
├── Parse Wasm module interface
├── Generate PureScript FFI declarations
├── Generate JS glue code
└── Handle common types (arrays, strings, records)
```

Inspired by: Rust's wasm-bindgen

Input (Rust):

```rust
#[wasm_bindgen]
pub fn step_simulation(
    positions: &mut [f32],
    velocities: &mut [f32],
    links: &[u32],
    alpha: f32
) -> f32 { ... }
```

Output (PureScript):

```
foreign import stepSimulation
  :: Float32Array
  -> Float32Array
  -> Uint32Array
```

```
  -> Number
  -> Effect Number
```

Phase 3: Numeric DSL with Wasm Backend

Goal: Write numeric code in PureScript, compile to Wasm.

```
-- User writes this in PureScript
forceCalculation :: V2 Number -> V2 Number -> V2 Number
forceCalculation p1 p2 =
  let delta = p2 `sub` p1
      dist = norm delta
      force = 1.0 / (dist * dist)
  in scale force (normalize delta)

-- Compiler generates efficient Wasm
-- No interpretive overhead
```

Approach: Finally-tagless with Wasm interpreter

```
class NumericDSL repr where
  lit :: Number -> repr Number
  add :: repr Number -> repr Number -> repr Number
  mul :: repr Number -> repr Number -> repr Number
  v2 :: repr Number -> repr Number -> repr (V2 Number)
  dot :: repr (V2 Number) -> repr (V2 Number) -> repr Number

-- Interpreter that generates Wasm bytecode
newtype WasmGen a = WasmGen (State WasmModule WasmExpr)

instance NumericDSL WasmGen where
  lit n = WasmGen $ pure (F32Const n)
  add a b = WasmGen $ do
    a' <- runWasmGen a
    b' <- runWasmGen b
    pure (F32Add a' b')
```

Phase 4: Build Tool Integration

Goal: `spago build` just works with Wasm dependencies.

```
# spago.yaml
package:
  name: my-viz
  dependencies:
    - psd3-simulation
  wasmDependencies:
```

```
    - name: force-kernel
      source: ./wasm/force-kernel
      bindings: auto  # Generate FFI automatically
```

```
$ spago build
Compiling PureScript...
Building Wasm dependencies...
  - force-kernel: cargo build --target wasm32-unknown-unknown
  - Generating FFI bindings...
Bundling...
Done.
```

Phase 5: Standard Numeric Library

Goal: Pre-built Wasm kernels for common operations.

```
import Numeric.Wasm.Linear (V2, V3, M44, dot, cross, matMul)
import Numeric.Wasm.Random (randomUniform, randomNormal)
import Numeric.Wasm.Stats (mean, variance, covariance)

-- These are thin PS wrappers over optimized Wasm
```

Existing Pieces to Build On

Component	Exists?	Notes
Force simulation in Rust	Many examples	d3-force algorithms well-documented
Rust → Wasm toolchain	Mature	wasm-pack, wasm-bindgen
PS FFI to JS	Mature	Works today
JS calling Wasm	Mature	WebAssembly API stable
Typed FFI generator	**GAP**	The key missing piece
Finally-tagless in PS	Exists (PSD3)	Foundation for DSL approach
SharedArrayBuffer	Available	For zero-copy data sharing

Alternative Demo Ideas

If force simulation doesn't resonate:

Demo	PS Strength	Wasm Strength
Real-time audio	Tidal pattern DSL	DSP, synthesis
Image processing	Filter pipeline composition	Convolution, transforms

Demo	PS Strength	Wasm Strength
Particle system	Declarative rules	Physics integration
Ray tracer	Scene description DSL	Ray-triangle intersection
Neural network	Type-safe layer composition	Matrix operations

The **Tidal connection** is particularly interesting—PSD3-Repos already has `purerl-tidal` and `purescript-psd3-tidal`. Real-time audio synthesis is exactly where Wasm shines.

Recommendations

Library Porting

1. **Start with `linear`**: Foundation for both diagrams and PSD3 improvements. Medium effort, high payoff.
2. **Add envelope/trace to PSD3**: Rather than full diagrams port, incorporate the key innovations into PSD3's layout system.
3. **Build `ad` incrementally**: Start with forward-mode (simpler), add reverse-mode for ML applications.
4. **Extend `purescript-machines`**: Build on existing Mealy machine foundation to add Source/Sink/Process abstractions.
5. **Coordinate with community**: Check PureScript Discord/forums for existing efforts before starting.

Infrastructure (The WASM Path)

6. **Build Phase 1 demo first**: Prove the PS + Wasm architecture works with manual FFI before investing in tooling.
7. **Develop `purescript-wasm-bindgen`**: The key missing piece—typed FFI generation from Wasm modules.
8. **Explore finally-tagless → Wasm compilation**: PSD3's expression system is the foundation; extend it with a Wasm interpreter.
9. **Consider Tidal as demo vehicle**: Real-time audio is a compelling use case where Wasm performance matters and PS's DSL capabilities shine.

Sources

PureScript Registry

- Pursuit Package Documentation
- PureScript Registry

Existing Packages

- purescript-linear-algebra
- purescript-gl-matrix
- purescript-js-bigints
- purescript-bigints
- purescript-dodo-printer
- purescript-optparse
- purescript-ssrs
- purescript-free
- purescript-decimals
- purescript-colors
- purescript-validation
- purescript-datetime
- purescript-unordered-collections
- purescript-density-codensity
- purescript-node-stream-pipes

Haskell Reference

- Diagrams Quick Start
- Diagrams User Manual
- diagrams-core on Hackage
- linear on Hackage
- ad on Hackage
- machines on Hackage

WebAssembly & Performance

- wasm-bindgen Guide
- wasm-bindgen GitHub
- purescript-wasm (Wasm bytecode generation)
- PureScript Discourse: WASM FFI
- purescript-machines
- Data.Machine.Mealy on Pursuit

Metaprogramming & DSLs

- Finally Tagless, Partially Evaluated (Okmij)
- JFP Paper (Carette, Kiselyov, Shan)

PSD3 WASM Implementation

- WASM Force Demo - Benchmark and three-way validation
- Kernel Separation Architecture - Pluggable kernel design
- PSD3 Simulation Library - WASMEngine implementation

Last updated: January 2026