PureScript REPL & Live Coding Environment Research

Status: active **Category**: research **Created**: 2026-01-29 **Author**: Claude (research session)

Executive Summary

This document captures research into improving PureScript's interactive development experience. The current PSCi REPL spawns a fresh Node process per expression, preventing state persistence. Rather than just fixing PSCi, we're exploring whether a Bret Victor-style live coding environment (like Haskell for Mac) might be a better goal.

Key insight: The goal isn't "better REPL" but "better feedback loop for understanding code."

Part 1: Current PSCi Architecture

The Core Problem

From `purescript/app/Command/REPL.hs:111-118`:

```
eval _ _ = do
  writeFile indexFile "import('./$PSCI/index.js').then(({ $main }) =>
$main());"
  result <- readNodeProcessWithExitCode nodePath (nodeArgs ++ [indexFile])
""
```

Each expression evaluation:

1. Compiles a temporary `$PSCI` module (all let bindings + new expression)
2. Generates JavaScript to `.psci_modules/$PSCI/index.js`
3. Spawns a **new Node process** to evaluate
4. Captures stdout, process terminates

Persists (Haskell memory): imports, let bindings as AST, externs **Doesn't persist** (JavaScript): runtime state, connections, mutable variables

Prior Improvement Attempts

1. **Browser Backend (PR #2199, 2016)** - Merged then removed in v0.15.0
2. **Fabrizio's Socket Server** - Haskell socket server + Node client, abandoned
3. **natefaubion's 2020 Collaboration** - Proposed persistent Node + richer eval protocol, stalled

psc-ide: The Right Architecture Already Exists

psc-ide (the IDE server) already has:

- Long-running Haskell process with TVar state
- Socket-based protocol

- Fast incremental rebuilds
- Type environment in memory

Missing only: JavaScript evaluation capability

Part 2: Haskell for Mac & Live Coding Environments

Haskell for Mac (Manuel Chakravarty)

Haskell for Mac provides an Xcode-like IDE with "Haskell Playgrounds" - the key innovation.

How Playgrounds Work:

- A playground is like an editor buffer where you type what you would usually type into GHCi
- When you change your program, it is **constantly executed and rerun**
- You see changes immediately as results update
- Results can be text, graphics, web pages, animations, or games
- Integrates Apple's SpriteKit for game development

Key difference from REPL: You're not typing commands sequentially. You're editing a document that continuously shows its computed output.

Architecture (from Haskell Interlude podcast #72):

- Swift frontend + Haskell backend via C FFI
- **Links GHC as a library** for interactive features
- Multi-modal output: text, HTML, graphics

Critical lesson learned: Tight coupling to GHC API caused maintenance burden. Every GHC version change required app updates. Advanced users couldn't choose GHC versions. This led to a successor project using **LSP instead of direct GHC integration**.

Implication for PureScript: Don't embed the compiler directly. Use a server (like Try PureScript or psc-ide) that can be updated independently.

Swift Playgrounds (Apple)

Swift Playgrounds has similar architecture with some documented details:

Execution Model:

- Results of operations presented in a step-by-step timeline as they execute
- Variables can be logged and inspected at any point
- By default, playgrounds are synchronous - each line evaluated imperatively
- `PlaygroundSupport` framework controls lifecycle

Key APIs:

- `PlaygroundPage.current.liveView` - slot for continuously rendered UI
- `needsIndefiniteExecution = true` - keeps playground running after top-level code completes
- `finishExecution()` - terminates the playground process

Communication: Uses NotificationCenter to communicate between playground and host IDE.

Light Table (Chris Granger)

Light Table was directly inspired by Bret Victor's "Inventing on Principle."

Core Ideas:

- **Inline evaluation**: Type `(+ 3 4)` and immediately see `7` - no ctrl-enter needed
- **Value flow visualization**: See how values flow through arbitrarily complex functions
- **Watchers**: Track values in real time across JavaScript, ClojureScript, Python
- **Live views**: Control a browser right inside the editor

Architecture:

- Built almost entirely in ClojureScript
- Uses node-webkit (now Electron)
- CodeMirror for editing
- BOT architecture (Behavior, Objects, Tag) for extensibility
- Modified Clojure compiler to retain position metadata

Key insight: "Files are not the best representation of code, just a convenient serialization."

Quokka.js

Quokka.js provides instant JavaScript/TypeScript execution in editors.

Features:

- Code runs immediately as you type, on unsaved changes
- Error messages displayed right next to causing code
- Console logs and values displayed inline
- **Value Explorer**: Tree view for navigating complex objects
- **Time Machine**: Move forward/backward through execution

Technology: Same instrumentation engine as Wallaby.js (commercial test runner). Years of R&D on JS code instrumentation.

Observable Notebooks

Observable takes a different approach - reactive dataflow.

Model:

- Notebooks execute in order of **data flow**, not linear sequence
- Like spreadsheets - changing one cell updates all dependent cells
- Reactive primitives: `viewof` and `mutable`
- Open-source runtime stitches cells into dependency graph

Key insight: Linear order of cells doesn't matter. The runtime figures out dependencies.

Part 3: Bret Victor's Principles

From Learnable Programming (2012):

What the Environment Must Show

1. **Vocabulary meanings**: Labels and context make code readable without memorization
2. **Program state and data**: "The entire purpose of code is to manipulate data, and we never see the data"
3. **Hidden state elimination**: Implicit state (like current fill color) must be visible

Victor's key claim: **"Show the data"** is more important than live coding.

Specific Recommended Features

For understanding vocabulary:

- Mouse-over labels for function parameters
- Context-aware annotations connecting code to output

For following execution flow:

- **Scrubbing sliders**: Move through execution step-by-step
- **Timeline visualization**: See which lines execute and when
- **Pattern visualization**: Loops and conditionals shown visually

For seeing state:

- Display all calculated values (numbers, colors, shape thumbnails)
- Automatic data visualization (tables → plots for large datasets)
- Visible state panels (current fill color, transform matrices)

For create-by-reacting workflow:

- Autocomplete with default arguments pre-filled showing immediate results
- Direct manipulation of output to adjust values (draggable numbers)

For create-by-abstracting:

- Seamless constant → variable conversion
- Interactive control showing effects in real-time
- Automated code transformations (variable → function argument, code → loop)

Victor's Critique

> "A live-coding Processing environment addresses neither of these goals. JavaScript and Processing are poorly-designed languages that support weak ways of thinking... And live coding, as a standalone feature, is worthless."

The point: Live coding alone isn't the goal. **Understanding** is the goal. Live coding is just one tool toward that.

Part 4: Existing PureScript Infrastructure to Build On

Try PureScript

Try PureScript already has much of what we need:

Server (Haskell/Scotty):

- Pre-compiles 200+ packages at startup
- Holds type environment and externs in memory
- Compiles single `Main` module per request (fast)
- Returns JavaScript + type errors/warnings

Client (Halogen):

- Ace editor with PureScript syntax
- Sandboxed iframe for execution
- Import path remapping to server URLs
- ES Module Shims for npm package resolution

Limitations:

- Stateless - each compile creates fresh iframe
- Full module compilation overhead
- No expression-level evaluation
- No state persistence between evaluations

purescript-language-server / psc-ide

Long-running process with the right architecture:

- TVar-based state management
- Socket protocol for commands
- Fast single-file rebuilds using cached externs
- Type environment always available

Could potentially be extended with evaluation capability.

IPurescript (Jupyter Kernel)

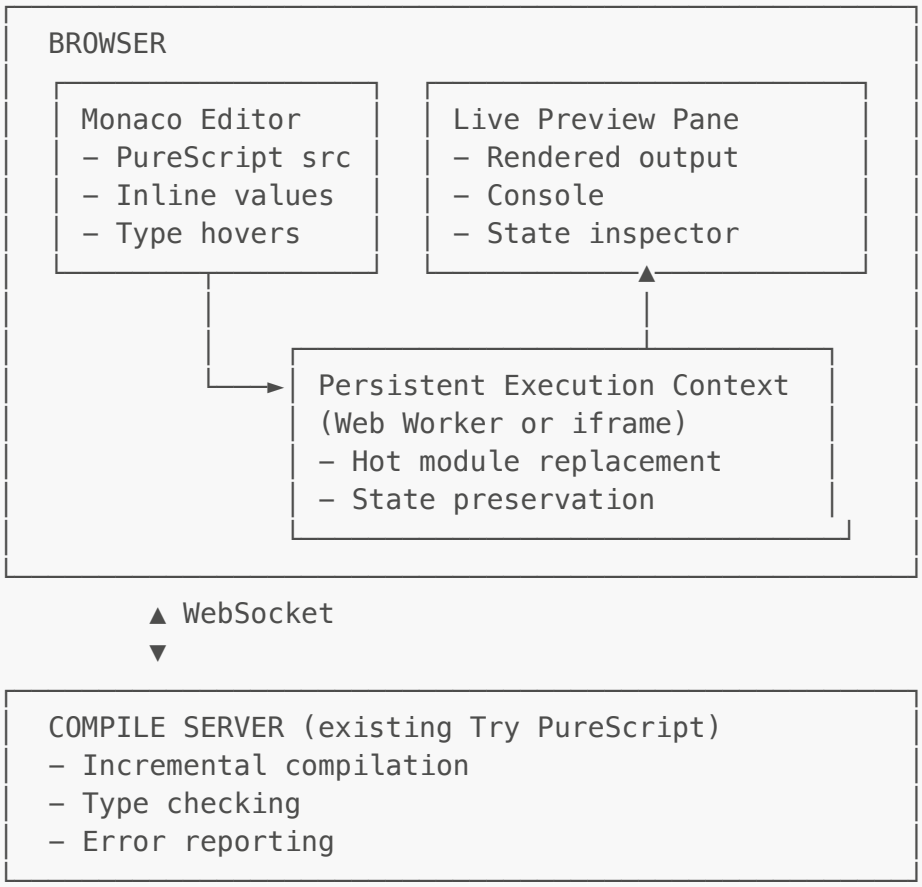
ipurescript - Jupyter kernel wrapper, work in progress/abandoned. Creates a temp PureScript project, shared among kernel instances.

Part 5: Implementation Options

Option A: Enhanced Try PureScript (Browser-Based)

Extend Try PureScript with playground-like features:

Architecture:



Key changes from current Try PureScript:

1. **Persistent iframe/worker** - Don't tear down between compiles
2. **Hot module replacement** - Swap compiled modules without full reload
3. **Inline value display** - Instrument code to report intermediate values
4. **State inspector** - Show current variable bindings

Pros:

- Builds on existing infrastructure
- No new binaries to distribute
- Works anywhere with a browser

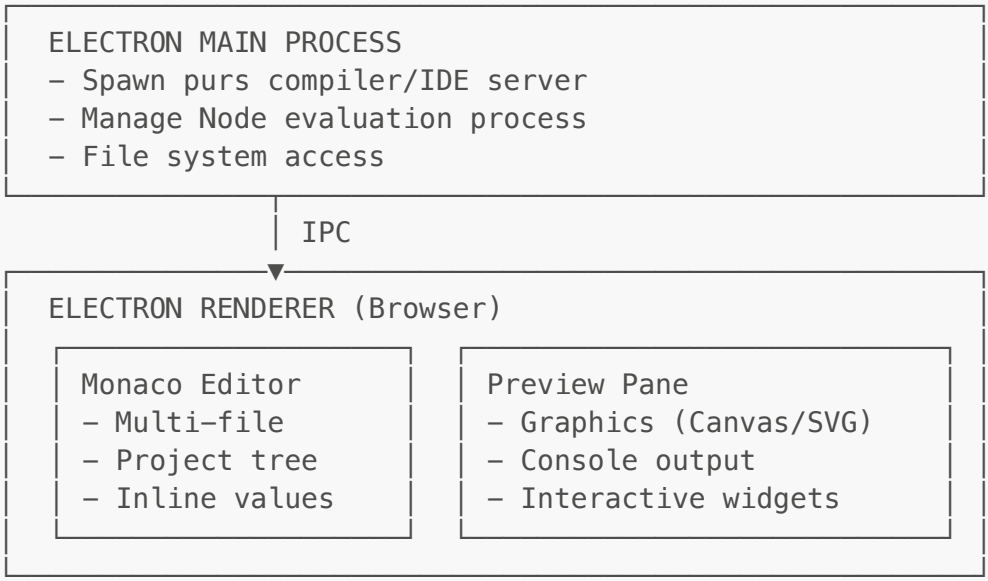
Cons:

- Limited to browser APIs
- Compile server needs to be running
- No filesystem access

Option B: Electron App (Haskell for Mac Style)

Full desktop application with embedded compiler:

Architecture:



Technology choices:

- **Electron**: Cross-platform, good for media-rich apps
- **Monaco Editor**: VSCode's editor, excellent PureScript support possible
- **purs ide server**: For compilation and type info
- **Persistent Node process**: For evaluation with state

Pros:

- Full system access (files, Node APIs)
- Can bundle compiler
- Richer graphics possibilities (Canvas, WebGL)
- Works offline

Cons:

- Larger distribution
- Need to maintain native app
- Cross-platform testing burden

Option C: VSCode Extension

Build as extension to existing editor:

Architecture:

- Use existing PureScript IDE extension for compilation
- Add custom "Playground" document type
- Webview panel for preview
- Extension host manages evaluation

Pros:

- Familiar environment for developers

- Leverage existing PureScript tooling
- Large existing user base

Cons:

- Constrained by VSCode extension API
- Can't customize editor deeply
- Webview communication overhead

Option D: Observable-Style Reactive Notebook

Different paradigm - reactive cells instead of imperative code:

```
-- Cell 1: Define data
data = [1, 2, 3, 4, 5]

-- Cell 2: Transform (auto-updates when Cell 1 changes)
doubled = map (_ * 2) data
-- → [2, 4, 6, 8, 10]

-- Cell 3: Visualize (auto-updates when Cell 2 changes)
chart = barChart doubled
-- → [rendered SVG]
```

Pros:

- Natural fit for data exploration
- Dependencies explicit
- Order doesn't matter

Cons:

- Different mental model from traditional coding
- Less suitable for complex applications
- Would need custom runtime

Part 6: Key Technical Challenges

Challenge 1: Fast Incremental Compilation

PureScript compiler is already fast, but for truly responsive feel:

- Need sub-100ms compilation for small changes
- psc-ide's rebuild is close but optimized for single files
- May need expression-level compilation (smaller than module)

Challenge 2: State Preservation Across Reloads

Options:

1. **Serialize/deserialize** - Save state, reload, restore (complex for closures)
2. **Hot module replacement** - Swap code, keep state (needs runtime support)
3. **Replay** - Re-execute from beginning with cached inputs (deterministic only)

PureScript's purity helps here - less hidden mutable state to preserve.

Challenge 3: Instrumenting Code for Value Display

To show intermediate values inline, need to:

1. Transform AST to insert logging/reporting
2. Correlate reported values with source positions
3. Display without disrupting code flow

Could instrument at:

- CoreFn level (after type checking)
- JavaScript output level (simpler but loses type info)

Challenge 4: Graphics and Interactivity

For Bret Victor-style visualization:

- Need Canvas/SVG rendering capability
- Animation frame loop for continuous updates
- Event handling for interactive manipulation

Browser environment provides this naturally.

Part 7: Recommended Approach

Phase 1: Enhanced Try PureScript (MVP)

Start with browser-based, extending Try PureScript:

1. **Persistent execution context** (don't tear down iframe)
2. **Simple state display** (show let bindings and their values)
3. **Auto-recompile on change** (debounced)
4. **Better error display** (inline in editor)

This validates the concept with minimal new infrastructure.

Phase 2: Rich Visualization

Add Bret Victor-style features:

1. **Inline value annotations** (instrument compiled code)
2. **Graphics pane** (Canvas-based, for PSD3 visualizations!)
3. **Time scrubbing** (for animations)
4. **Direct manipulation** (click values to edit)

Phase 3: Desktop App (Optional)

If browser limitations become blocking:

1. **Electron wrapper** around Phase 1-2 work
2. **Local compiler** (bundle purs binary)
3. **File system integration**
4. **Offline support**

Resources

PureScript REPL Issues

- PSCi Remote Connection - Issue #2142
- Async in PSCi - Issue #2218
- REPL Improvement Discourse
- psc-ide Design Doc

Live Coding Environments

- Haskell for Mac
- Light Table
- Quokka.js
- Observable
- Swift Playgrounds Support

Bret Victor

- Learnable Programming
- Inventing on Principle (talk)

Implementation References

- Try PureScript
- Monaco Editor
- ESM-HMR
- IPurescript Jupyter Kernel
- purescript-halogen-monaco - Halogen bindings for Monaco
- purescript-ace-halogen - Halogen bindings for Ace

Part 8: PSD3 Integration Opportunity

Why This Matters for PSD3

PSD3 is a visualization library. Visualizations are fundamentally about **seeing data**. A playground environment that shows visualization output as code changes would be the ideal development experience for PSD3 users.

Current workflow for PSD3 development:

1. Edit code
2. Run `spago bundle`
3. Refresh browser
4. See if visualization looks right
5. Repeat

Ideal workflow:

1. Edit code
2. See visualization update immediately

What a PSD3 Playground Could Offer

Code Panel (Monaco/Ace):

- PureScript source
- Type information on hover
- Error highlighting

Visualization Panel (Canvas/SVG):

- Live-rendered D3/SVG output
- Animation playback
- Scrubbing through force simulation frames

State Inspector:

- Current simulation state
- Node/link data
- Force parameters

Direct Manipulation:

- Drag nodes in visualization → see position values update
- Adjust force parameters with sliders → see simulation change

This would be a showcase application for PSD3 while also being a useful development tool.

ForcePlayground as Proof of Concept

The existing `ForcePlayground` demo (`purescript-hylograph-simulation/demo`) already demonstrates much of what we'd want:

Current ForcePlayground features:

- Live-updating force-directed graph visualization
- Interactive force configuration (toggle charge, collide, link, x, y forces)
- Preset layouts (Centered, Floating, Quadrants)
- Category filtering with animated transitions
- Drag interaction for node repositioning

- Phyllotaxis animations to quadrant positions

What a playground could add:

- **Code pane** showing the PureScript source alongside the visualization
- **Parameter tweaking** - edit `Simple.initialSetup` and see immediate changes
- **Value inspection** - hover over `currentNodes` to see actual simulation state
- **Time scrubbing** - pause simulation, scrub through tick history

The ForcePlayground is already ~700 lines of well-structured Halogen code. A playground environment could let users experiment with modifications without the rebuild cycle.

Technical Synergies

1. **psd3-selection HATS AST** - The Hylomorphic Abstract Tree Syntax already describes visualizations declaratively. A playground could show both the HATS tree and its rendered output.
2. **Browser-native execution** - PSD3 targets the browser anyway. No Node process needed for evaluation - just run the JS directly.
3. **Halogen infrastructure** - Try PureScript is already Halogen. PSD3 has Halogen integration (`psd3-simulation-halogen`). Same tech stack.
4. **Existing demos** - The `site/website` demo infrastructure could evolve into this playground.

Part 9: Concrete Next Steps

Immediate (Proof of Concept)

1. **Fork Try PureScript** or build minimal equivalent
2. **Replace Ace with Monaco** (better extensibility)
3. **Add persistent iframe** - Don't tear down between compiles
4. **Add visualization pane** - Render Canvas/SVG output
5. **Test with psd3-simulation** examples

Short-term (Usable Tool)

1. **HMR for ES modules** - Swap compiled modules without full reload
2. **Error mapping** - Show compilation errors inline in editor
3. **Type hovers** - Query psc-ide for type information
4. **Console capture** - Show `log` output in side panel

Medium-term (Bret Victor Features)

1. **Value annotations** - Instrument code to show intermediate values
2. **Time scrubbing** - For animations and simulations
3. **Direct manipulation** - Click values to edit, drag visual elements
4. **State persistence** - Preserve let bindings across recompiles

Architecture Decision: Browser vs Electron

Start with browser-based because:

- PSD3 targets browser anyway
- Try PureScript provides starting point
- No distribution/install friction
- Can always wrap in Electron later

Consider Electron if:

- Need local file system access
- Want to bundle compiler for offline use
- Need richer system integration

Part 10: Technical Implementation Notes

HMR Strategy for PureScript

Since PureScript is pure, state preservation is easier than typical JS HMR:

Option A: Full Replay

1. On code change, recompile
2. Re-execute entire program from scratch
3. Fast enough for most PSD3 visualizations

Option B: Selective Update

1. Track which modules changed
2. Only re-execute changed modules + dependents
3. Preserve unchanged state

Option C: Hybrid (recommended)

1. Default to full replay (simple, correct)
2. For long-running simulations, serialize state before reload
3. Resume simulation from serialized state

Instrumentation for Value Display

To show intermediate values inline:

```
-- Original code
result = map (_ * 2) [1, 2, 3]

-- Instrumented (at JS level)
const _line_1_result = (() => {
  const _v = map(x => x * 2)([1, 2, 3]);
  __report__(1, "result", _v);
  return _v;
})();
```

The `__report__` function sends value + position to the host environment.

Could also instrument at CoreFn level for richer type information.

Editor Integration

Try PureScript's `Editor.purs` provides a clean pattern:

- Halogen component wraps Ace
- Queries for get/set content, markers, annotations
- Output signals text changes
- Debounced (750ms) to avoid thrashing compiler

Same pattern works for Monaco via `purescript-halogen-monaco`.

Compile Server Options

Option 1: Try PureScript Server

- Existing infrastructure
- Pre-compiled packages
- WebSocket/HTTP API
- Requires server running

Option 2: Browser-based Compiler

- purs WASM build (if it existed)
- Fully offline
- Larger initial download

Option 3: Local purs + WebSocket Bridge

- Run `purs ide server` locally
- WebSocket bridge in browser
- Best for Electron app

For MVP, Option 1 (fork/adapt Try PureScript) is fastest path.