Typed Feedback Loops: From REPL to Reactive DAG

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

The Thread

This document synthesizes research from three directions that converge on the same insight:

1. **REPL Research** - Why PSCi is weak, what Haskell for Mac got right
2. **Bret Victor's Principles** - "Show the data" over "live coding"
3. **shaped-steer Vision** - Typed DAGs as the unifying abstraction

The convergent insight: The value isn't in a better REPL. It's in **typed reactive computation with immediate feedback** - where the type system constrains what's possible and immediate execution shows what actually happens.

Part 1: What's Wrong with REPLs

The PSCi Problem (Technical)

Each expression spawns a fresh Node process. State doesn't persist. But this is a symptom, not the disease.

The Deeper Problem (Conceptual)

REPLs are **linear**. You type, execute, see result, type again. This doesn't match how understanding develops:

```
REPL mental model:
  → type expression
  → see result
  → realize mistake
  → retype entire thing with fix
  → see new result
  → repeat
```

What you actually want:

```
Reactive model:
  → write expression
  → see result
  → tweak one part
  → see how result changes
  → tweak another part
  → understand the relationship
```

The difference: In a REPL, each command is independent. In a reactive system, expressions have **continuous existence** - they persist, and changes propagate.

What Haskell for Mac Got Right

From the Haskell Interlude podcast:

> "A playground is like an editor buffer where you type what you would usually type into GHCi. But when you change your program, it will always be executed and rerun all the time."

The key shift: **From command-response to continuous evaluation**.

You're not "asking the computer" to evaluate something. You're maintaining a document that continuously shows its computed meaning.

Part 2: Bret Victor's Actual Point

Victor's "Learnable Programming" essay is often cited for "live coding." But his actual thesis is sharper:

> "The entire purpose of code is to manipulate data, and we never see the data. We write with blindfolds."

Live coding alone is worthless without:

1. Showing the data at every stage
2. Making the connection between code and data explicit
3. Letting you manipulate either one (code or data) and see the other update

Victor's hierarchy:

1. **See the state** (most important)
2. **See how state changes over time**
3. **See the relationship between code and state**
4. Live execution is just the mechanism enabling these

Implication for PureScript

PureScript's purity is an advantage here. In imperative code, "the state" is scattered across mutable variables, closures, globals. In pure code, data flows explicitly through function composition. The state at any point IS the function's output.

A typed pure language is *more amenable* to Victor's vision than JavaScript/Processing.

Part 3: shaped-steer's Insight

The `UNIFIED-DAG-VISION.md` document crystallizes something profound:

> "Unix's power came from a simple universal abstraction (byte streams) plus a composition mechanism (pipes). The abstraction was intentionally impoverished."

> "Typed DAGs could be the next-level primitive. Not bytes flowing linearly, but nodes with typed edges, branching, merging, and crucially - memoization and incremental recomputation as built-in semantics."

The Unification

Tool	Surface	Reality
Spreadsheet	2D grid	DAG of formula dependencies
Notebook	Linear cells	DAG (incorrectly linearized)
Build system	Targets + rules	DAG of file dependencies
Playground	Code + output	DAG (code → result)

These are all the same structure. The "playground" isn't a new thing - it's seeing the DAG that was always there.

Finally-Tagless as the Key

From `UNIFIED-DATA-DSL.md`:

```
class DataDSL (repr :: Type -> Type) where
  source  :: forall a. DataSource a -> repr (Array a)
  mapA    :: forall a b. (a -> b) -> repr (Array a) -> repr (Array b)
  filterA :: forall a. (a -> Boolean) -> repr (Array a) -> repr (Array a)
```

The same expression, different interpreters:

Interpreter	Output
`SheetEval`	Actual computed values
`Deps`	Set of dependencies
`Pretty`	Readable formula string
`TypeCheck`	Type of result
`D3Eval`	Visualization DOM

This is exactly what a playground needs: write code once, interpret it multiple ways (execute, visualize, show types, track dependencies).

Part 4: The Convergent Design

What a "PureScript Playground" Actually Is

Not a REPL. Not an IDE. It's:

A reactive DAG editor where nodes are typed PureScript expressions, edges are data dependencies, and multiple synchronized views show the same computation from different angles.

```
┌─────────────────────────────────────────────────────────────────────┐
│                        THE DOCUMENT                                  │
│                                                                      │
│  x = [1, 2, 3, 4, 5]                      x : Array Int             │
│                                            [1, 2, 3, 4, 5]           │
│                                                                      │
│  doubled = map (_ * 2) x                  doubled : Array Int       │
│                                            [2, 4, 6, 8, 10]          │
│                                                                      │
│  big = filter (_ > 5) doubled             big : Array Int           │
│                                            [6, 8, 10]                │
│                                                                      │
├─────────────────────────────────────────────────────────────────────┤
│  DEPENDENCY VIEW:  x ──→ doubled ──→ big                            │
├─────────────────────────────────────────────────────────────────────┤
│  TYPE VIEW:  Array Int → Array Int → Array Int                      │
└─────────────────────────────────────────────────────────────────────┘
```

Change `x` to `[1, 5, 10]`. Watch `doubled` become `[2, 10, 20]`, `big` become `[10, 20]`. Immediately.

The Type System as Guide

This is where PureScript shines. When you write:

```
y = filter (_ > "5") x  -- x : Array Int
```

You don't wait until runtime to discover the error. The playground shows:

```
y = filter (_ > "5") x
           ^^^^^^^^
    Error: Cannot compare Int with String

    (_ > "5") has type String → Boolean
    but filter expects Int → Boolean
```

The type system is **part of the feedback loop**. It's not a gate you pass to run code - it's continuous guidance showing what's well-formed.

Why Dependent Types Matter Here

With refinement types (Liquid Haskell style), you could write:

```
-- positives : Array { v : Int | v > 0 }
positives = filter (_ > 0) x

-- This would be a type error, not a runtime error:
head positives  -- Safe! Type system knows array is non-empty after filter
```

The feedback loop extends to **properties**, not just types. "Is this array non-empty?" becomes a type-level question the playground can answer before execution.

Part 5: Learning Applicatives Through Feedback

The user's comment:

> "i certainly feel like i might NEVER have understood applicatives and monads without that tool"

Here's how the playground teaches applicatives:

```
-- Step 1: What does pure do?
a = pure 3
-- a : Maybe Int
-- Just 3

-- Step 2: What about <*>?
b = pure (+) <*> Just 3 <*> Just 4
-- b : Maybe Int
-- Just 7

-- Step 3: What if one is Nothing?
c = pure (+) <*> Just 3 <*> Nothing
-- c : Maybe Int
-- Nothing

-- Step 4: What about lists?
d = pure (+) <*> [1,2] <*> [10,20]
-- d : Array Int
-- [11, 21, 12, 22]   -- All combinations!
```

You don't need to understand the theory first. You **play**. Change `Just 3` to `Just 10`. Change `Maybe` to `Array`. Watch what happens. The pattern emerges from observation.

The type signature is always visible. When you see that `<*>` has type `f (a → b) → f a → f b`, and you see the concrete behavior, the abstraction clicks.

Part 6: shaped-steer as the Implementation Path

The `TYPED-CELLS-DESIGN.md` document already describes the architecture:

Server: Extended Try PureScript

```
POST /compile-expr
{
  "expr": "\\xs -> filter (_ > 0) xs",
  "context": {
    "imports": ["Data.Array"],
    "localTypes": {}
  }
}

Response:
{
  "js": "(xs) => Data_Array.filter(x => x > 0)(xs)",
  "type": "Array Int -> Array Int",
  "deps": ["Data.Array"]
}
```

Client: Halogen with Multiple Views

```
data CellOutput
  = ValueOutput Value       -- The computed result
  | TypeError CompileError  -- What went wrong
  | TypeInfo String         -- The inferred type
  | DepGraph (Set CellId)   -- What this depends on
```

Execution: Reactive DAG Evaluation

```
evaluateNotebook :: Notebook -> Aff Notebook
evaluateNotebook notebook = do
  let deps = buildCellDeps notebook.cells
  let order = topologicalSort deps
  foldM evaluateCellInContext notebook.initialContext order
```

This IS the playground architecture. The notebook mode with typed cells IS the feedback loop we've been describing.

Part 7: The Minimal Valuable Product

Not shaped-steer (too ambitious for MVP)

The full vision involves Excel import, Makefile parsing, database persistence, AI co-driving. That's years of work.

Not Try PureScript (too simple)

Try PureScript compiles a single Main module and runs it. No persistent bindings, no dependency tracking, no inline type display.

The Sweet Spot: Typed Playground

A browser-based environment that:

1. **Parses a document** into top-level bindings
2. **Compiles each binding** via Try PureScript server
3. **Builds dependency graph** from references
4. **Evaluates in topological order**
5. **Shows type + value** for each binding
6. **Recompiles on change** (debounced)
7. **Highlights errors inline** with type information

That's it. No grid mode. No Makefile import. No AI. Just typed reactive evaluation.

Implementation Estimate

Given Try PureScript exists:

- `/compile-expr` endpoint: 1-2 sessions
- Client document parser: 1 session
- Dependency graph + ordering: 1 session
- Evaluation loop: 1 session
- UI (Monaco + results pane): 2 sessions
- Polish + error handling: 2 sessions

~8-10 sessions to MVP that demonstrates the core value.

Part 8: Connection to Liquid PureScript (Next Research)

The typed feedback loop becomes even more powerful with refinement types:

```
-- Current PureScript
head :: forall a. Array a -> Maybe a  -- Might be Nothing

-- With refinements
head :: forall a. { v : Array a | length v > 0 } -> a  -- Always succeeds
```

The playground could show:

- When refinements are satisfied
- What constraints flow from operations
- Which values can flow to which positions

This turns the type system from "accepts or rejects" into "shows what properties hold at each point."

Research question: How feasible is adding Liquid Haskell-style refinement types to PureScript?

Summary

Approach	What You Get	What's Missing
PSCi	Execute expressions	State, types visible, feedback
Try PureScript	Compile + run	Multiple bindings, persistence
Observable	Reactive dataflow	Types, PureScript
shaped-steer	Everything	Exists as vision, not implementation
Typed Playground	Core feedback loop	Graphics, AI, Excel import

The path forward:

1. **Build Typed Playground** as proof of concept
2. Validate that the feedback loop delivers learning value
3. Evolve toward shaped-steer vision as features prove out

References

- `docs/kb/research/purescript-repl-research.md` - REPL research
- `apps/shaped-steer/docs/UNIFIED-DAG-VISION.md` - DAG vision
- `apps/shaped-steer/docs/TYPED-CELLS-DESIGN.md` - Implementation design
- `apps/shaped-steer/docs/UNIFIED-DATA-DSL.md` - Finally-tagless DSL
- Learnable Programming - Bret Victor
- Haskell Interlude #72 - Manuel Chakravarty on playgrounds