Preview

Shipping a Production Web App in Elm

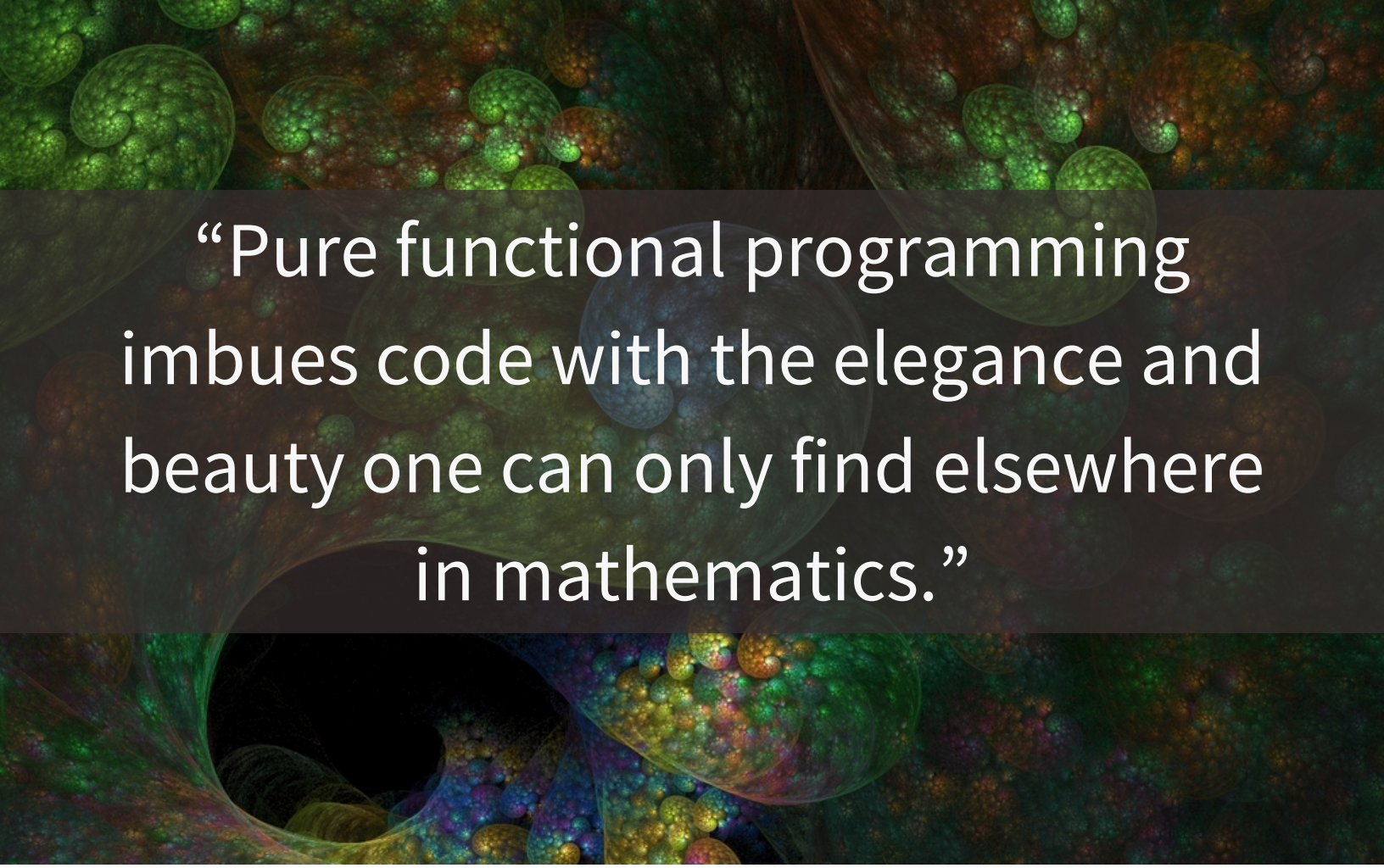
Welcome! Let's start with a quick survey.

order these 3 by your expected happiness if you were coding in them 40 hours per week

HASKELL

JAVA

PYTHON

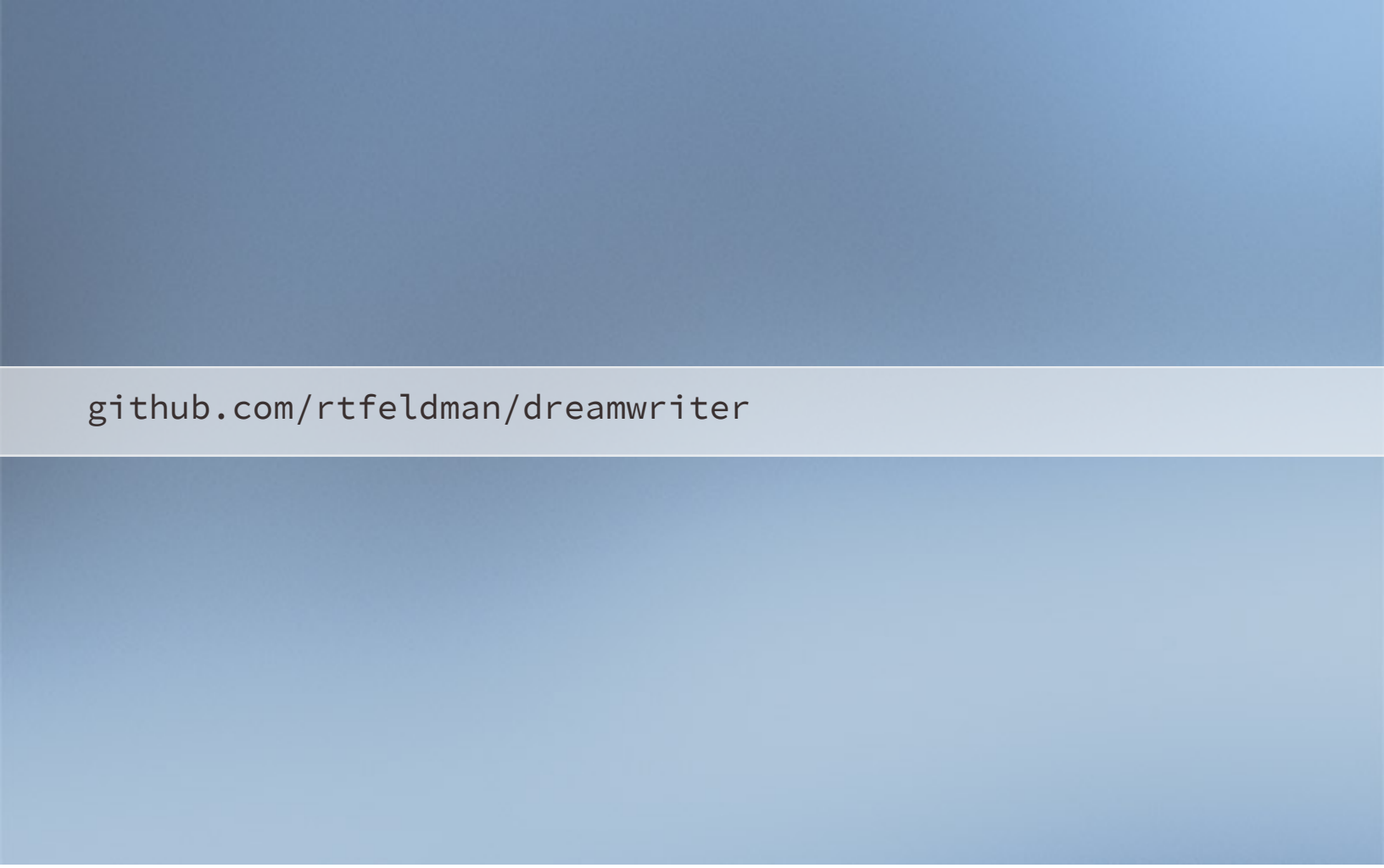


This quote embodies what draws many to functional programming.

That's not what draws me to it. I like to build things—specifically, I like to build great user experiences. They often grow complicated...

...and, as they grow, have seen them become increasingly prone to collapse.

I arrived at functional programming because I wanted the things I built to stop breaking.



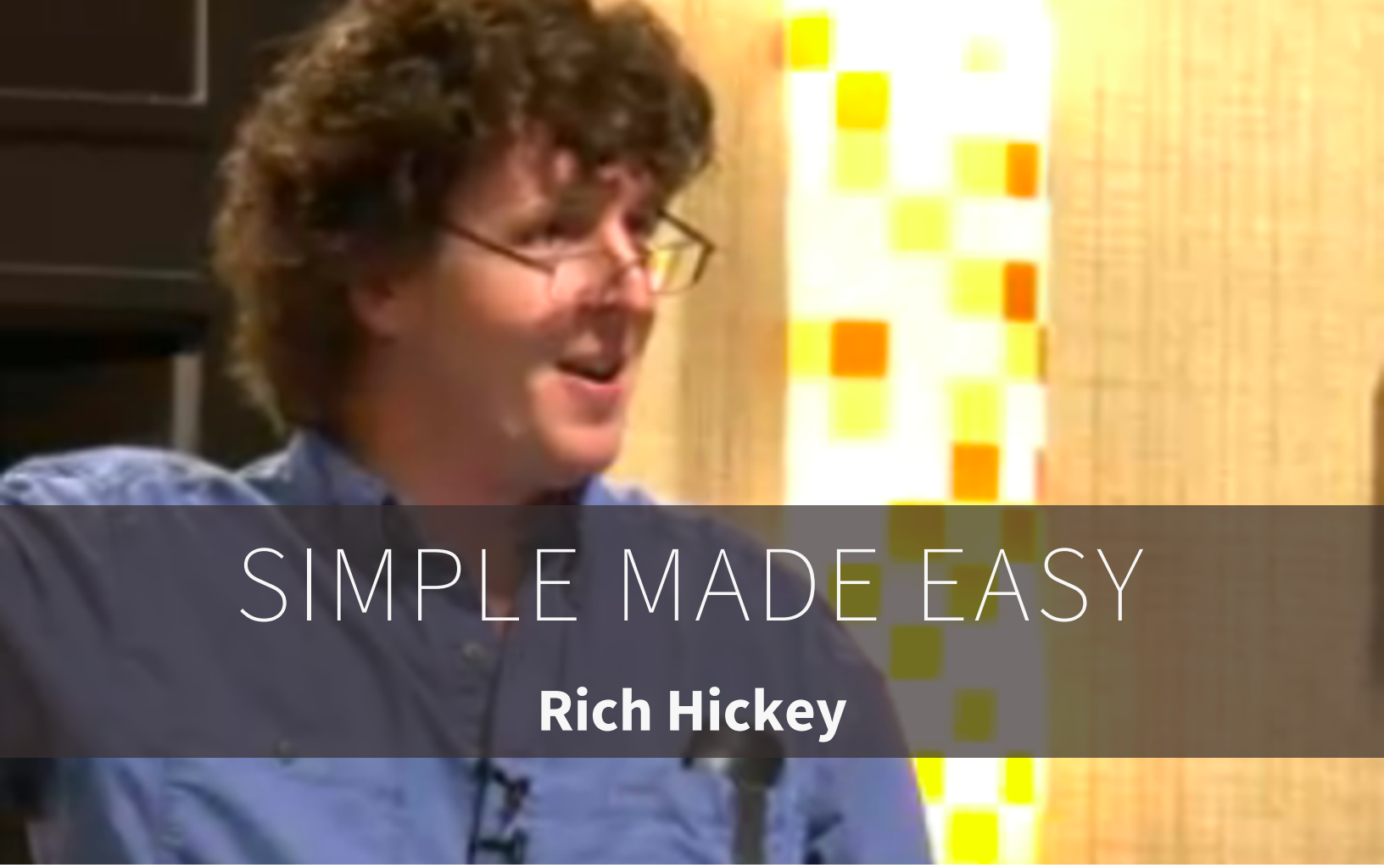
Here's one example. I created Dreamwriter (http://dreamwriter.io) to use for myself.

Original Architecture:

GIMOJ

The original architecture was GIMOJ.

It didn't start out as a mess, but ended up there after several iterations. It eventually got bad enough that a rewrite was clearly the only way out.



Between when I'd started the project and realized I needed to rewrite, I saw this talk.

Simple is an objective notion.

— RICH HICKEY

One of the most powerful ideas from this talk is that simplicity is objective; no matter who you are, no matter how easy or difficult it is for you personally to understand or to use it, a particular concept has the same amount of simplicity or complexity.

Simple: Less interleaving of concepts

Easy: Near to your current skill set

According to Rich Hickey's definitions, in the context of programming, Simple is about less interleaving of concepts whereas Easy is about being close to your current skill set.

SIMPLER CODE

is more maintainable!

There's a limit to what the human mind can process at once, and simpler programs are more maintainable because you are less prone to exceeding that limit.

Over the course of Rich Hickey's talk, I became convinced that I had been underrating simple code. I wanted to try emphasizing simplicity in my rewrite.

I thought "okay, this is interesting...go on..."

STATELESS FUNCTIONS

minimize interleaving

Stateless (aka pure, aka referentially transparent) functions minimize interleaving by excluding the possibility of side effects. As such, they are intrinsically decoupled from one another.

Mutation, side effects, and reading from shared state all serve to interleave logic that could potentially be decoupled if expressed in a different way.

FUNCTIONAL STYLE

in an imperative language

So I started moving in that direction. I tried reach for mutation less often, and to write more stateless functions.

REACT.JS

...and Flux!

When React.js came out, adopting it was another step in the right direction. It let me change over even more of my logic to a more functional style.

GOOD PERFORMANCE
SIMPLE RENDER LOGIC
TAME LEARNING CURVE

Adopting React reinforced for me the idea that things could be better across the board. It didn't take long to pick up, it had good performance, and it made my rendering logic much simpler. It really delivered.

At this point I was like "okay, this functional programming thing is definitely making my life easier...now how can I take it a step further?"

When you’re looking for something simple, you want to see it have focus.

— RICH HICKEY

Focus is about saying “no.”

— STEVE JOBS

Why is this? Why is functional code simpler?

React was a good start, but I wanted

LESS

Discipline is

HARD

Invariants are

EASY

OVER 200 LANGUAGES

compile to JavaScript

Over 200 languages compile to JavaScript...and that's just from reading the list on CoffeeScript's GitHub repo. I'm sure there are even more out there that haven't made the list. I had to narrow it down.

DEAL-BREAKERS

slow compiled JS
poor interop with JS libraries
unlikely to develop a community

I had three primary deal-breakers that disqualified the overwhelming majority of these languages.

Slow compiled JS means a poor UX, and I won't tolerate that. Anything with bloated JS that is going to drag down performance is right out.

Poor interop means I'll expect to reinvent a lot of wheels. If there's nothing available in the altJS language, I want to be able to fall back on the native JS implementation that someone else has already written and battle-tested.

Community is critical to any language. Community means a rich library ecosystem. It means you can use StackOverflow to quickly find solutions to your questions. It means you can find other programmers capable of contributing to your project right now. Many of these languages are clearly someone's pet project, and are extremely unlikely to develop a flourishing community - ever.

JS, BUT SUCKS LESS

Dart

TypeScript

CoffeeScript

These three are the most popular options among altJS languages that embrace JavaScript's semantics.

The code you write in these languages is intended to be very similar to the same code you'd write in JavaScript, but the experience is nicer.

The two main upsides to this approach are (1) a shallow learning curve, and (2) that interop with JS tends to be trivial.

EASILY TALKS TO JS

Elm

PureScript

ClojureScript

Among languages that massively deviate JS semantics, interop is always the big question.

How easy is it to leverage the massive ecosystem of JS libraries in this language? (Put another way: how many wheels will I be expected to reinvent?) How efficient is the compiled JS code? To what degree can I use raw JS to implement performance optimizations where appropriate?

Among the alternatives I considered, these three seemed to have the best interop stories.

COFFEESCRIPT

CLOJURESCRIPT

PURESCRIPT

CLOJURESCRIPT: PROS

fairly mature

flourishing community

shallow learning curve

CLOJURESCRIPT: CONS

immutability is not an invariant

any function can have side effects

the $1B mistake

Cons (get it? Cons?)

Immutability is the default, but it's not an invariant. Some values are mutable, so I still have to remember which ones are which.

PURESCRIPT: PROS

100% immutability, type inference

JS interop: just add type signature

functions cannot have side effects*

PURESCRIPT: CONS

potential for lost invariants via FFI

least mature

steep learning curve

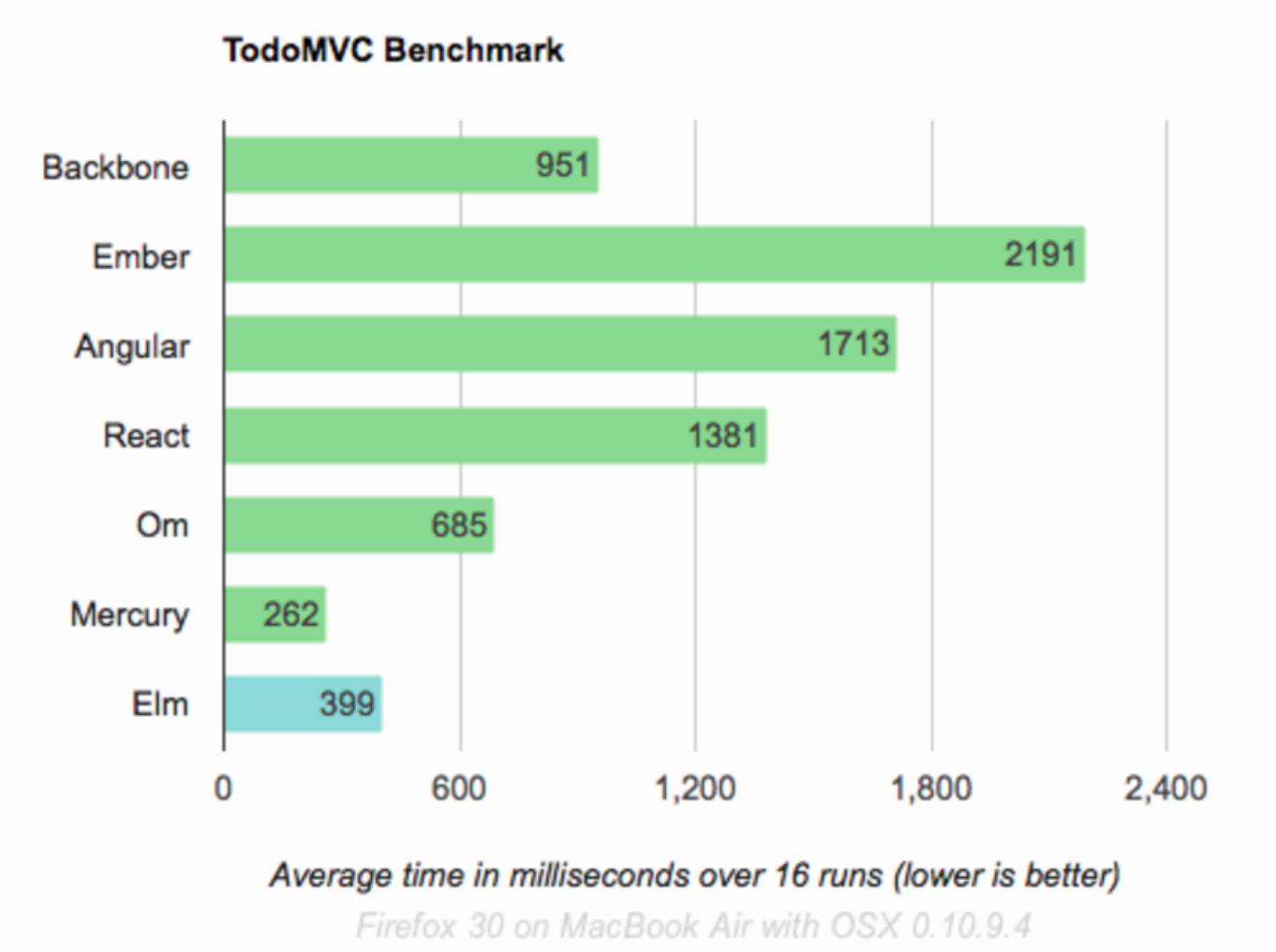
Particularly scary was the idea of adopting a purely functional language. How would I get anything done without side effects?

REWRITE IN PURESCRIPT

Only one way to find out!

...but there was no working React analog at the time. There was some preliminary stuff, but nothing that was ready for prime time.

Around this time, a blog post came out...



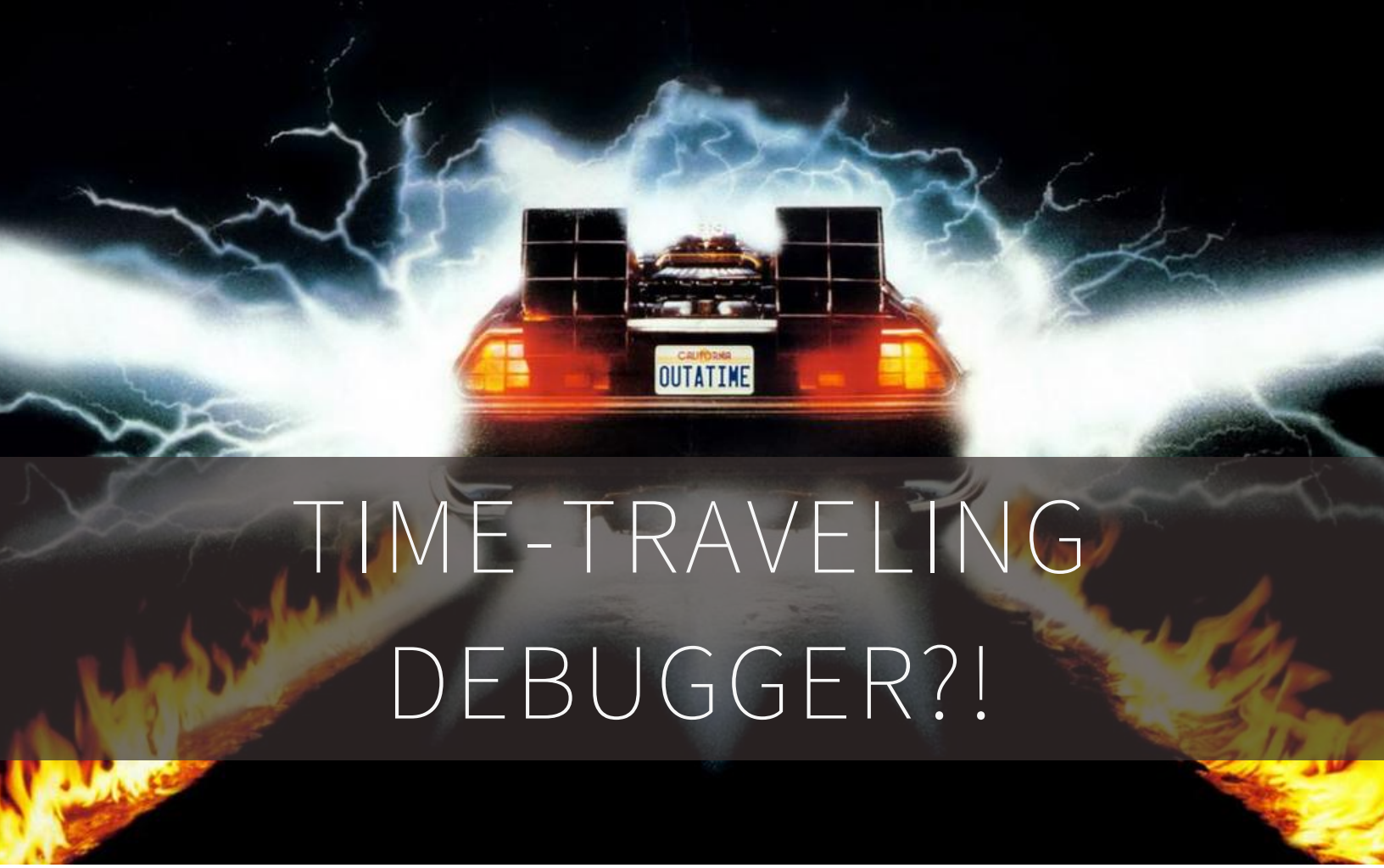
"Blazing Fast HTML in Elm" - this blog post described how Elm could now be used to do React-style virtual DOM rendering.

JS INTEROP: PORTS

Elm/JS relationship is like client/server

Only raw data allowed in and out

Pub/sub communication system



TIME-TRAVELING DEBUGGER?!

ELM: PROS

100% type inference & immutability

functions cannot have side effects

best support for best architecture

Look familiar?

"Functions cannot have side effects" doesn't have an asterisk here. It's really not possible, even accidentally, even if you're doing interop.

The best architecture I've ever encountered for programming applications is basically what React + Flux does, and Elm's language primitives embrace that in a big way. Architecting an Elm application in what I've found to be "the best way" feels very natural and well-supported. It's harder in every other language I've used.

ELM: CONS

most verbose JS interop

no way to “cheat” invariants

no effects sequencing (in those days)

Here we go!

So I started with the Giant Mess of JQuery.

Goal Architecture:

PURELY FUNCTIONAL ELM

I wanted to end up with a purely functional Elm code base.

Intermediate Architecture:

FUNCTIONAL-STYLE COFFEESCRIPT

(React + Flux)

Along the way I had this rewrite in functional-style CoffeeScript, centered around React and Flux.

ANTICIPATE INVARIANTS

Use stateless functions wherever possible.

The key to making this a smooth transition was to anticipate the invariants I'd have once I translated this CoffeeScript/React/Flux code to Elm. I could port all the stateless functions over one at a time in isolation, often as direct one-to-one translations.

CoffeeScript/React/Flux Rearchitect

github.com/rtfeldman/dreamwriter-coffee/tree/strangeloop

Elm Rewrite

github.com/rtfeldman/dreamwriter/tree/strangeloop

```
div [class "sidebar"] [
  div [class "sidebar-header"] [
    input [placeholder "search notes",
      onInput searchNotes targetValue] [],

    span [id "new-note-button",
      onClick newNote ()] []
  ],
  div [class "sidebar-body"] [sidebarBody],
  sidebarFooter
]
```

```
input [placeholder "search notes",
       onInput searchNotes targetValue] []
```

onInput

```
.addEventListener("input")
```

```
input [placeholder "search notes",
       onInput searchNotes targetValue] []
```

targetValue

```
function(event) {
  return event.target.value;
}
```

```
input [placeholder "search notes",
       onInput searchNotes targetValue] []
```

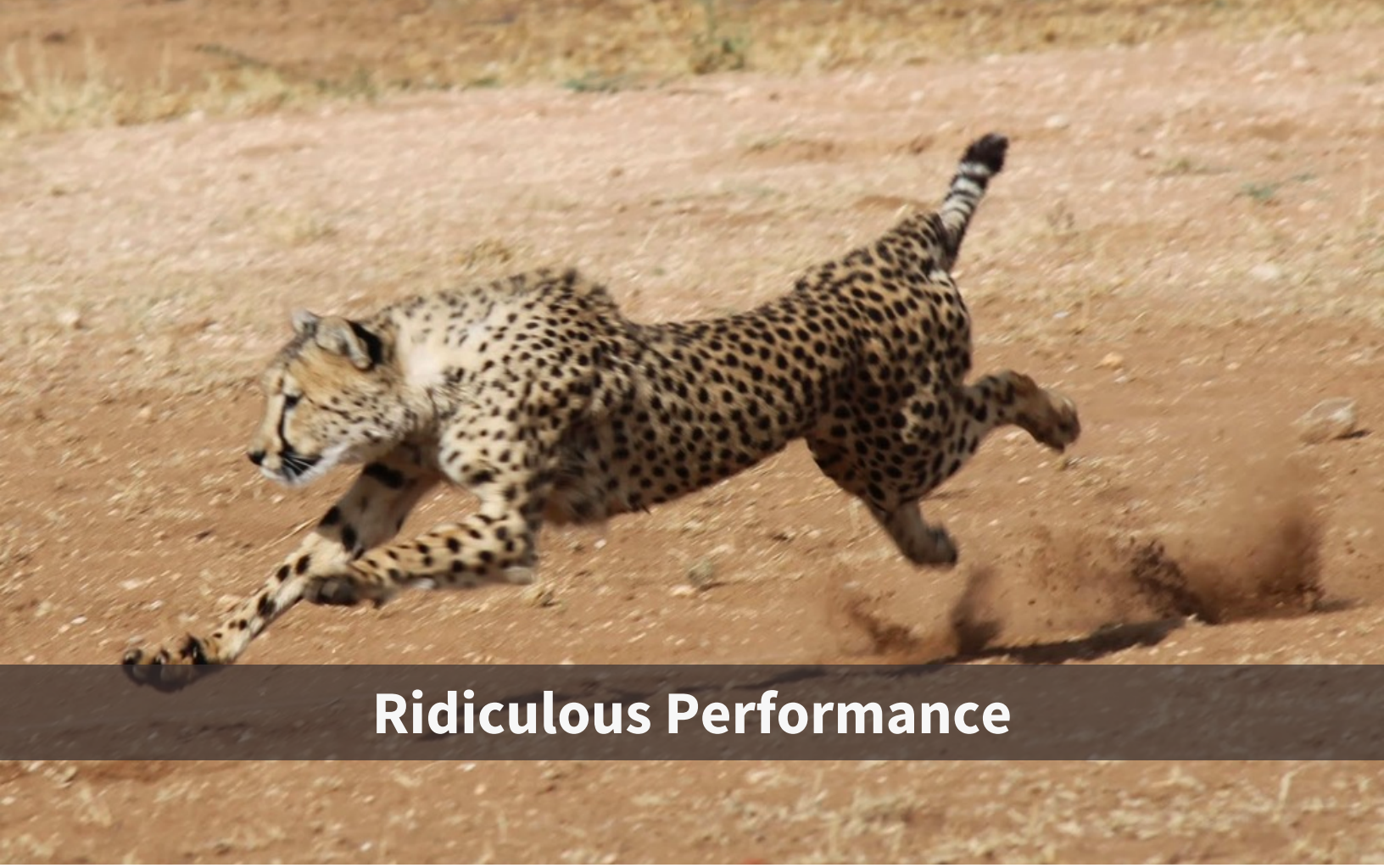
searchNotes

```
function(queryString) {
  return {
    actionType: "SEARCH_NOTES",
    payload: queryString
  };
}
```

```
view model openNote =
  case model.currentNote of
    Nothing ->
      viewNotes model.searchResults openNote

    Just note ->
      viewSingleNote note
```

SO HOW WAS IT?



Ridiculous Performance

LANGUAGE USABILITY

```
-- TYPE MISMATCH ------------------------------------

Not all elements of this list are the same type of value.

I noticed the mismatch in element #4, but go through
and make sure every element is the same type of value.

42|
43|   [10,9,8,"7"]
44|           ^^^

To be more specific, as I infer all the types, I am seeing
a conflict between this type:
```

The next release of Elm, which will be out in a week or two, is going to include language usability improvements like these tasty error messages. I've really come to appreciate that Elm values things like this, in the same way that I appreciate when applications have a nice UX.


Refactoring is THE MOST FUN THING

SEMANTIC VERSIONING GUARANTEED

For every single package.

Automatically enforced.

Yes, really.


No Runtime Exceptions

Of all the languages I've ever used,

ELM IS THE SIMPLEST.

This is why. Maintaining and improving Dreamwriter in production has been an absolute blast. I'm never going back!

No Runtime Exceptions

Semantic Versioning Guaranteed

Time-Traveling Debugger

ELM-LANG.ORG

WOULD I LIKE TO SHIP ANOTHER WEB APP IN ELM?

Oh yeah!!!

Thanks!