SHIPPING A PRODUCTION WEB APP IN ELM

@rtfeldman

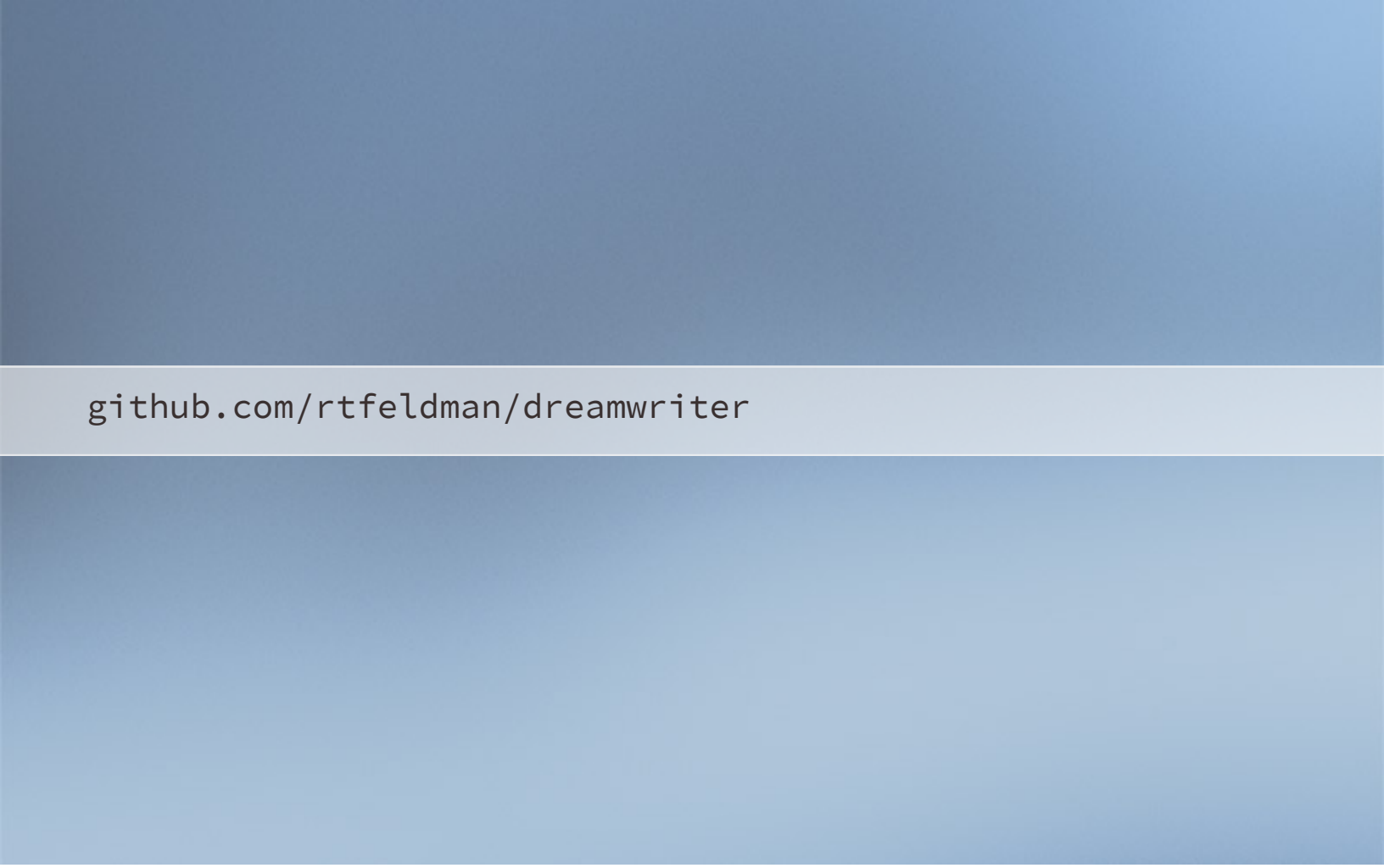
For the latest version of these, visit
https://presentate.com/rtfeldman/talks/shipping-a-...

One of the interesting things about a conference like LambdaConf is finding out all the different paths people take to functional programming. A love of mathematics and abstractions are common paths.

As a programmer, I like to build things—specifically, I like to build great user experiences. As they grow and I refine them, they become more intricate, which makes problems more likely to manifest. I've found it's all too easy to aim for building something like this...

...only to end up with something like this instead.

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



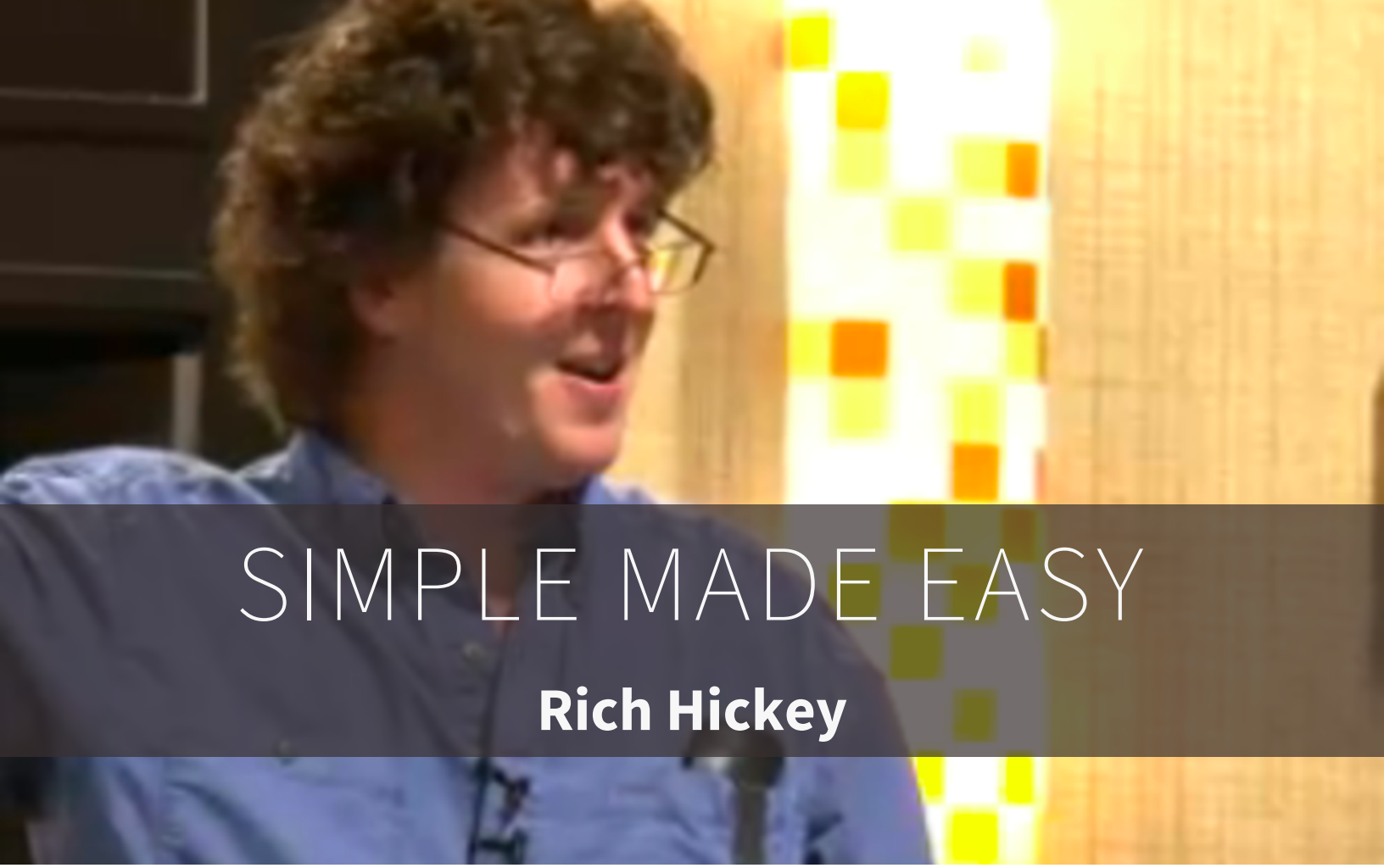
Here's one example. I created Dreamwriter (http://dreamwriter.io) to use for myself.

Initial 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

nearer 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.

FAMILIARITY GROWS OVER TIME

...but complexity is forever

You can become familiar with new concepts over time, but you can't learn your way out of complexity. All you can do is take steps to reduce it.

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 of Dreamwriter.

I was intrigued. How could I make my code simpler?

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 took me one step further in this direction. It let me change over even more of my logic to a more functional style.

React 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 thinking "okay, this functional programming thing is definitely making my life easier...now how can I take it a step further?" In other words, how could I get more of this?

Discipline is

HARD

Invariants are

EASY

One remaining source of bugs came from when my self-enforced invariants broke down. I'd accidentally mutate something, or introduce a side effect without realizing it, or maybe a third-party library would do the same. These bugs were often time-consuming to track down.

OVER 200 LANGUAGES

compile to JavaScript

I knew there were languages that would let me replace discipline with enforced invariants, so I could stop worrying about these things. But which language to choose? There were so many languages that compile to JavaScript...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 LESS PAINFUL

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.

Still, they didn't offer anything in the way of immutability or eliminating side effects. On those fronts, I'd still need just as much disicpline as before.

EASILY TALKS TO JS

Elm

PureScript

ClojureScript

Among languages that massively deviate from 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

I'd initially written off Elm because it seemed to go a different direction from CSS - either on Canvas or on the DOM - and I wanted to reuse all the CSS I'd already written for Dreamwriter.

CLOJURESCRIPT

flourishing community

trivial JS interop

side effects and mutation allowed

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

PURESCRIPT

100% immutability, type inference

JS interop: just add type signature

functions cannot have side effects*

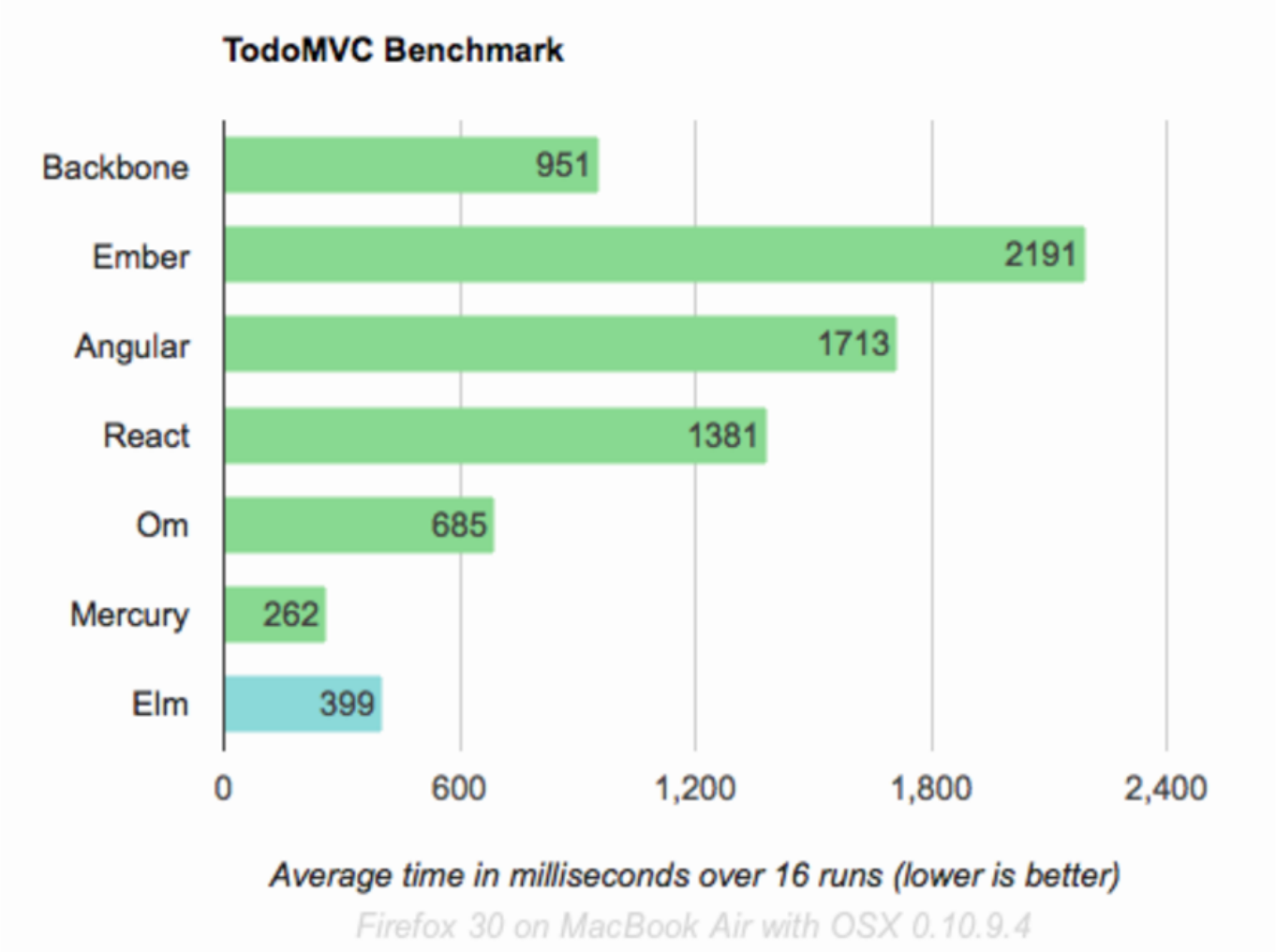
PureScript is designed for 100% immutability and no side effects, although it's technically possible for mutation and side effects to sneak in through FFI.

REWRITE IN PURESCRIPT

I had a winner!

...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

I looked further into Elm's JS interop and found that it was a bit more involved than what ClojureScript and PureScript did, but the result was that it actually maintained the invariants that everything in Elm Land was immutable and had no side effects.

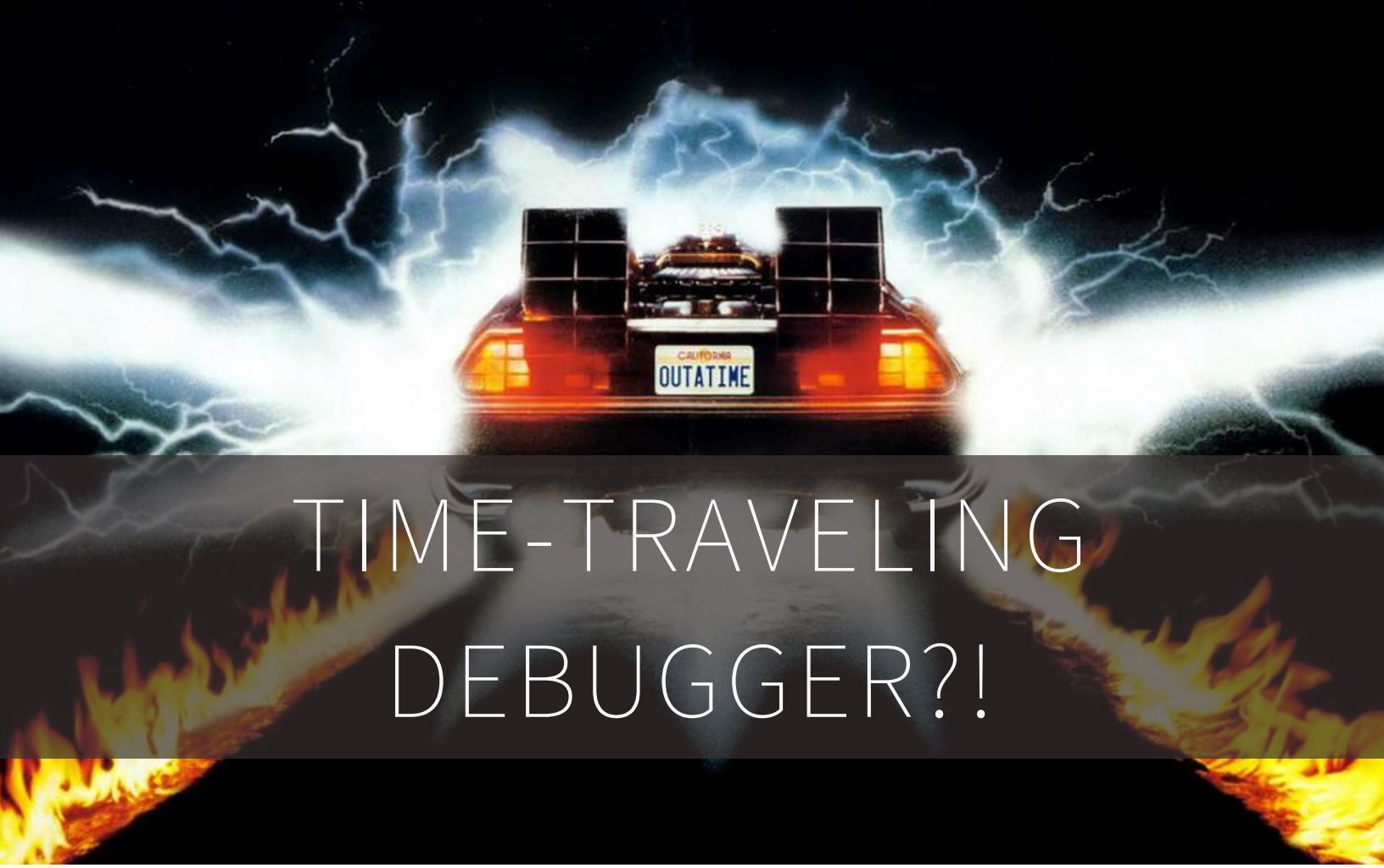
ELM

100% immutability, type inference

JS interop preserves invariants

functions have no side effects

In short, here was a language that had everything I needed: good performance, good JS interop, React/Flux-style rendering, and nothing but stateless functions and immutability...anywhere!



TIME-TRAVELING DEBUGGER?!

Then I saw this: the time-traveling debugger. http://debug.elm-lang.org/edit/Mario.elm

I was sold. 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 partial 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 was able to 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`

If you're interested, I tagged both the CoffeeScript/React/Flux Rearchitect and the Elm Rewrite when they were at feature parity. So you can take a look at both code bases, which produce pretty much the same application, and compare how they do things.

```
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
]
```

Here's some Elm code. It describes some nested divs with classes, and an input and span nested within those. Let's zoom in on that "onInput" part.

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

onInput

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

This roughly translates (with some hand-waving) to a JavaScript event listener for the "input" event, which will fire whenever the user types in the search text box, or does a Cut, Paste, etc. answer.

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

targetValue

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

Next let's look at targetValue. This is simply a function that translates the input event's `event` object into `event.target.value`, which will be the string the user entered into the text box.

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

searchNotes

```
function(queryString) {
  return {
    action: "SEARCH_NOTES",
    payload: queryString,
    destination: actionSignal
  };
}
```

Finally we have searchNotes, which accepts the string from the event that represents what the user wants to search for. Notice that it simply looks at its arguments and returns data - a representation of what the user has done. This JavaScript code is not *exactly* how it looks in Elm, but it gives you the general idea.

The "action" and "payload" fields here represent the action the user took, and the associated data. The destination refers to the Signal the action will be sent to; Elm programs are built up of Signals, which you can read more about at http://elm-lang.org/learn/Using-Signals.elm

```
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
]
```

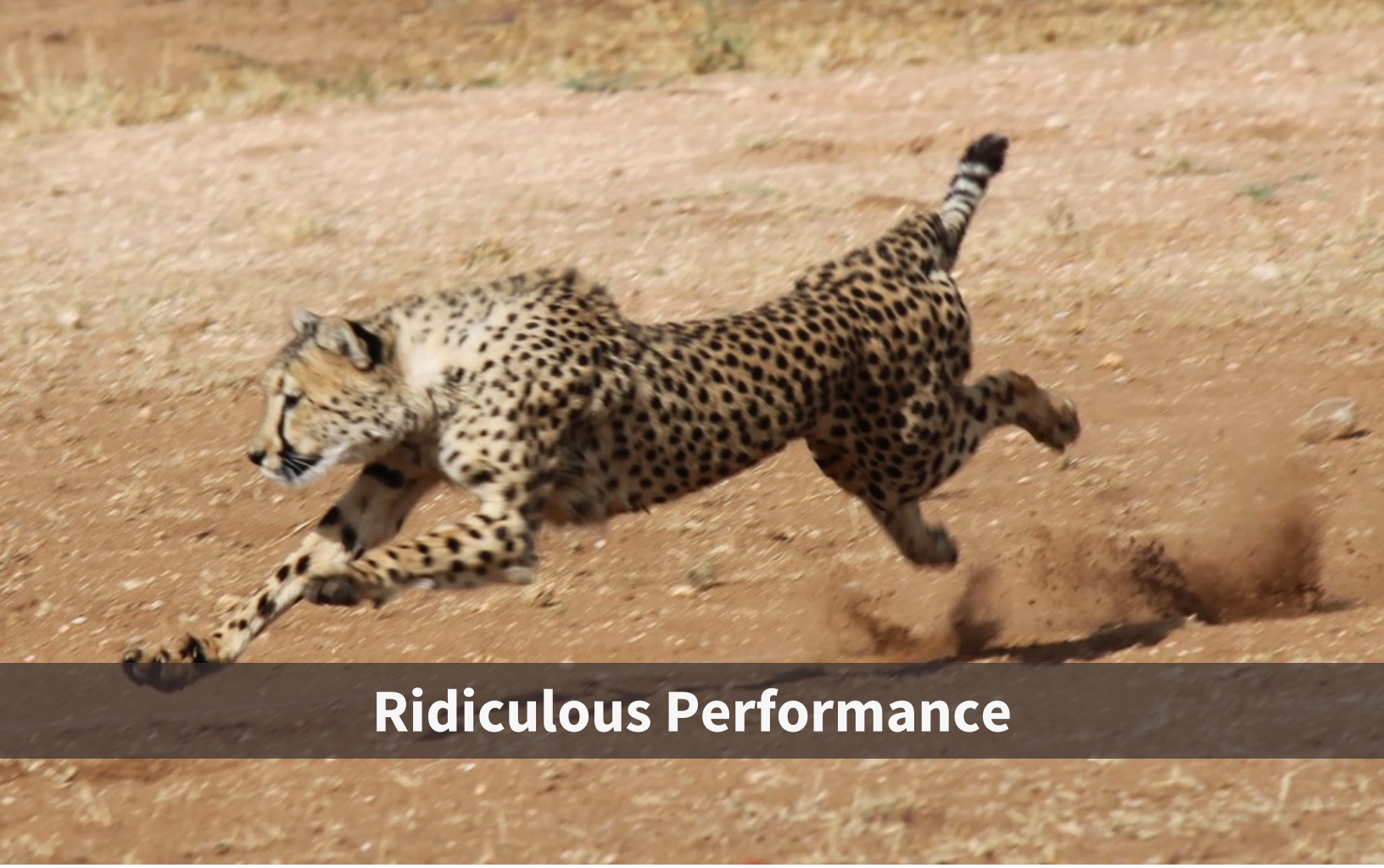
One cool thing about this approach - of using stateless functions and describing what actions will go to which signals using only immutable data - is that you get complete control over which code paths can have which effects on application state.

For example, that newNote function might incorporate a destination signal that allows new notes to be created, whereas searchNotes might incorporate one that allows actions to result in a notes search.

For debugging purposes, a setup like this would mean it really would be impossible for the New Note button to impact searching notes in any way, and for Search Notes to impact New Note in any way. Since there are no side effects, their possible impact on the program is limited to whichever signals you pass them!

How was the overall experience of shipping a production web app in Elm?

It was amazing!



Ridiculous Performance

The performance was great, and optimizing it as the code base grew was a breeze.

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.

I keep having this experience where I make a major breaking refactor, recompile, fix all the compiler errors...and when I bring up Dreamwriter, everything still works! No regressions! This experience makes refactoring SO much fun!

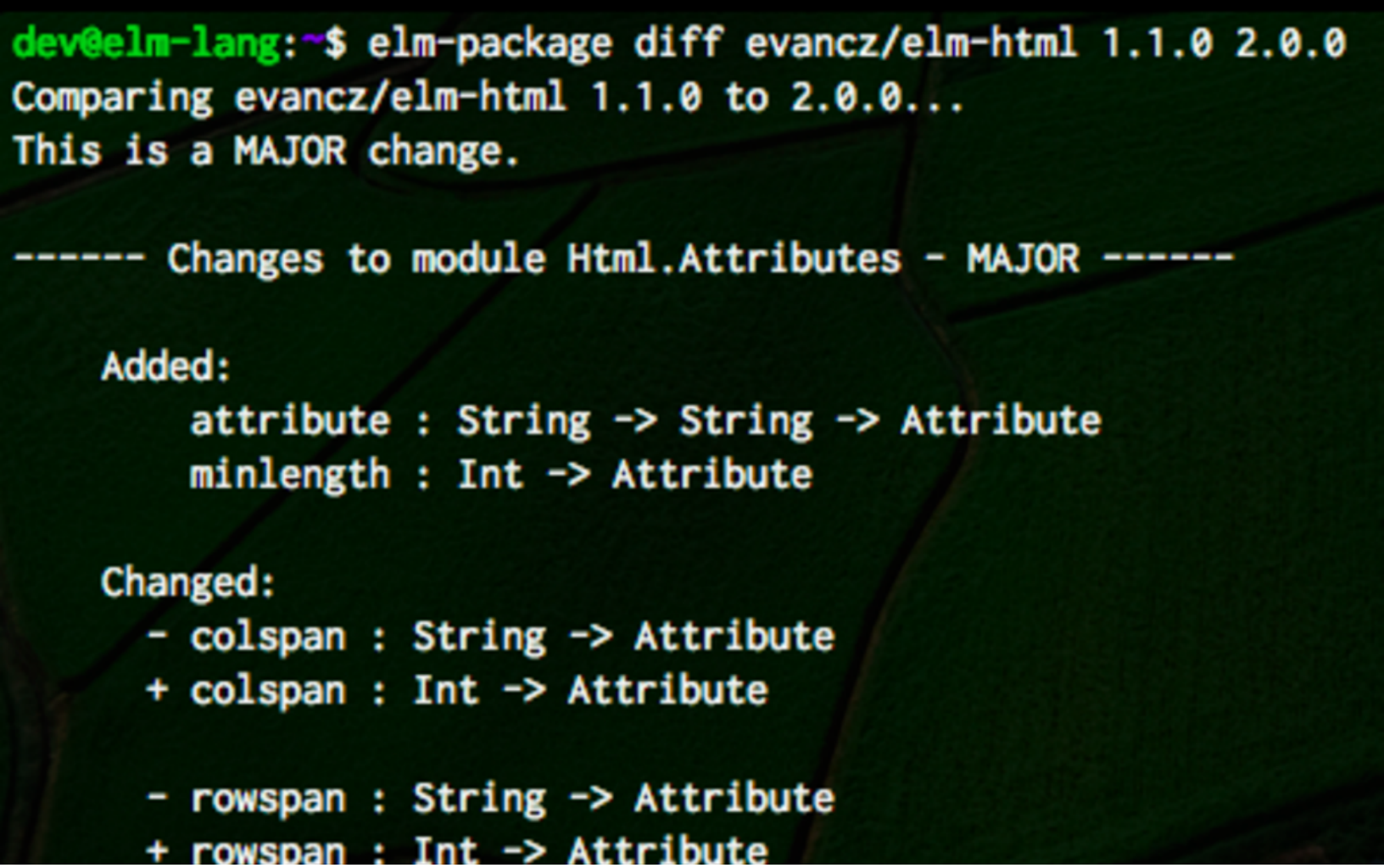
SEMANTIC VERSIONING GUARANTEED

For every single package.

Automatically enforced.

Yes, really.

The package manager is wonderful.



You can even run `elm-package diff` to tell you what has changed between two versions of a package, so you can know what you're getting yourself into before you upgrade.

No Runtime Exceptions

I still have yet to get a single runtime exception from my Elm code. The compiler really is that good! As someone who is used to seeing stuff like "undefined is not a function" over and over every day, this has been a huge breath of fresh air.

In short, shipping a production web app in Elm has been wonderful.

Now I finally feel I've found a language that lets me build awesome things that don't fall over, but rather which stand the test of time.

Of all the languages I've ever used,

ELM IS THE SIMPLEST.

I think the fundamental reason for this great experience is how simple Elm is.

ELM

stateless functions

immutable data

NO MESS.

This simplicity has made maintaining and improving Dreamwriter in production an absolute blast. I never want to go back!

No Runtime Exceptions

Semantic Versioning Guaranteed

Time-Traveling Debugger

ELM-LANG.ORG

Check out http://elm-lang.org or the mailing list at https://groups.google.com/forum/#!forum/elm-discuss for more!

WOULD I LIKE TO SHIP ANOTHER WEB APP IN ELM?

Would I do it all again, now that I'm familiar with Elm?

Oh yeah!!!

Thanks!