Beyond JavaScript: Building Stable Web Applications

JACK SWIGGETT, Stanford University

JavaScript is the dominant language in web application development, and allows for the creation of powerful and cross-platform applications. However, as a dynamically typed language with a constantly expanding array of features and a knack for concealing errors, JavaScript is often hard to maintain in large applications. In this paper I explore two programming languages, Elm and Reason, that provide an alternative to JavaScript for writing web applications. Both benefit from advanced type systems and functional design paradigms that seek to reduce bugs and improve code quality. The two languages embrace a largely similar approach to web application design, but each has advantages and disadvantages in terms of language design, framework design, usefulness of errors, and available tooling and documentation. I created the same simple web application using both languages, and here I compare my experience with each throughout the development process.

CCS Concepts: • **Software and its engineering → Functional languages**; **Domain specific languages**; *Data types and structures*; *Software libraries and repositories*;

Additional Key Words and Phrases: Web development, Elm, Reason, React

1 BACKGROUND

Most modern web applications are written using JavaScript. As a dynamically typed, interpreted language, JavaScript makes it easy to write code quickly and explore different ideas, without needing to plan out the entire structure of an application. However, this flexibility comes with drawbacks. It is often difficult to tell how data is being passed around in web applications, and new developers may have a hard time understanding an existing codebase. When errors do arise, it is difficult to pinpoint and resolve them.

One approach to tackling this problem is to gradually introduce static types into JavaScript. TypeScript and Flow are two projects that try to do this. Both projects embrace JavaScript backwards-compatibility, meaning that it is easy to take an existing JavaScript codebase and annotate it over time in order to increase type safety. This is a selling point for teams with lots of existing code, and it means that developers can iterate quickly in early prototypes, without worrying about types until later. However, it also means that lazy developers can easily omit types, or that third-party libraries may not provide them. As a result, it is very uncommon for projects created in TypeScript or Flow to have full type coverage, and type errors can still occur.

Because of this, many web developers are moving towards another approach: write in an entirely different language that inherently enforces type safety and code maintainability. Then compile from that language into JavaScript so that the code can be run in a browser. Elm and Reason are two languages built for that purpose.

1.1 Elm

Elm is a programming language that was designed by Evan Czaplicki in 2012. It is purpose-built for creating web applications, with the intent of eliminating runtime errors and enforcing best practices. Elm is a purely functional language, inspired by other functional languages including Haskell and OCaml. Elm code compiles directly to JavaScript.

1.2 Reason

Reason is actually a JavaScript-like syntax for OCaml, a functional programming language that has existed for over twenty years. Reason is developed in partnership with BuckleScript, a separate project that provides a backend for the OCaml compiler to compile OCaml into JavaScript. Compiling Reason code involves first converting it into OCaml syntax, and then compiling that into JavaScript using BuckleScript. For this project I also used ReasonReact, a library that allows developers to write ReactJS code using Reason.

1.3 Project Goals

This project comprised two primary goals. First, I wanted to become familiar with writing web applications using functional languages, and write a simple web application using both Elm and Reason. Before this quarter, I had never used a functional programming language, much less written a web application in one. Second, I wanted to compare the development experience using both languages. I wanted to compare various aspects of language design, features, and tooling, in order to determine which of these two options would be a better choice for designing web applications in the future. In that same vein, I wanted to compare writing code in Elm and Reason to my previous experience developing web applications in JavaScript, to see whether the benefits of shifting to a functional language outweigh the drawbacks and justify the learning curve.

2 APPROACH

I built visually and functionally equivalent versions of the same web application, using both Elm and Reason. I started from scratch in both languages, and only made use of standard libraries and, in the case of Reason, the ReasonReact library. Throughout the development process, I took notes on the language features I used, the errors I received, and my subjective assessment of the process for developing in each language.

2.1 Application

The best way to have a realistic development experience is to build an application that you actually want to use. I also wanted to build something eccentric enough that I would be unlikely to find it as a starting example for either language, but that would use a variety of web application features. I do swing dancing for fun, so I built an app to catalog and search swing dance moves. It has the following features:

- Store a list of swing dance moves (the "catalog"). Each move comprises a name, a list of tags describing that move, and a list of URLs to online videos of people doing or teaching that move.
- Allow the user to search the catalog by name or by a given tag.
- Allow the user to specify a new name, list of tags, and list of URLs, and add that move to the catalog.
- Allow the user to delete any move in the catalog.

2.2 Process

To begin, I read documentation and completed introductory tutorials for both Elm and Reason, as well as ReasonReact, in order to get a feeling for both languages/frameworks and how they are used to create web applications. Then I designed a simple user interface for the web application I wanted to build using Sketch. I duplicated this same interface using vanilla HTML and CSS, in order to render a static version of the website in a browser.

Once this was complete, I created Elm and Reason projects that rendered that same static webpage (without any user interactivity). Since both the Elm and Reason versions of the application are based off the initial HTML/CSS implementation, they both create the exact same HTML elements in the DOM and use the same `style.css` file. I then created types in both languages to model the structure of the application data (i.e. swing dance moves). Finally, I began working on the brunt of the development process: rendering moves and user interface elements based on an underlying model, and allowing users to search, create, and delete moves.

2.3 Code

The HTML/CSS, Elm, and Reason code, as well as the compiled JavaScript output from Elm and Reason, is available on GitHub at **https://github.com/jackswiggett/functional-web-apps**. Instructions for running each web application are included there.

3 RESULTS: COMPARING ELM AND REASON

3.1 Getting Started

Both Elm and Reason were easy to get up and running. I was installing on Mac OS X. Elm provides an OS X installation package, while Reason simply requires installing a global `npm` package. In addition, both languages provide plugins for most major editors, including Emacs, Vim, Sublime Text, Atom, and VS Code. Elm and Reason also provide command line utilities that make it easy to set up new projects and begin compiling and viewing code.

Both languages also offer a wide variety of tutorials and beginner guides to bring new developers up to speed. However, Elm provides an easy-to-use Read-Eval-Print Loop (REPL), whereas Reason does not. Even though OCaml has this functionality, it has not been ported to Reason. I found the Elm REPL very useful when learning the basics of language syntax, exploring the types of various expressions, etc., and I wished I had the same tool for Reason. However, both projects provide online tools to write and compile code in the browser, so there is still an easy way to try out simple Reason programs.

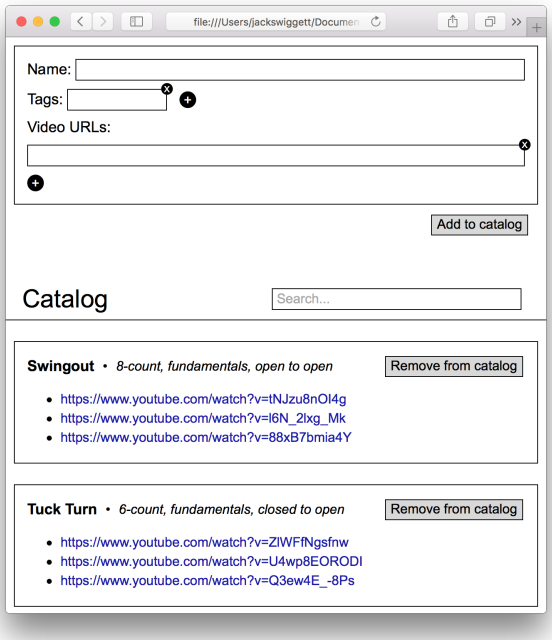

Fig. 1. The application user interface. The versions in pure HTML, Elm, and Reason all look exactly the same.

3.2 Modeling Application Data

Both Elm and ReasonReact adopt an approach to modeling application data built around actions and reducers. This approach is also commonly used in ReactJS applications with libraries like Redux. The state of the application (in this case, the moves in the catalog and the contents of user input boxes) is stored as a record, potentially with many fields and other nested records. Rather than modifying this state directly, the application must trigger specific actions—for example, `RemoveFromCatalogById`—which can include associated data, e.g. the id of the move that should be removed from the catalog. Any important user interactions, such as clicking buttons or editing input boxes, trigger actions. Those actions are then processed by a reducer function (called "reducer" in Reason and "update" in Elm). That function receives the current state and the action, and returns a new record containing the new state. It can make arbitrary changes to the application state, but they must be deterministic, i.e. a certain combination of input state and action must always lead to the same output state.

This approach has many advantages. If your application is in an unexpected state, it is easy to trace any actions that have occurred to identify which one modified the state incorrectly. In addition, it is possible to test the user interface independently from the application logic, by manually setting the state and then rendering the webpage. By itself, this provides a great advantage over vanilla JavaScript applications, although libraries like Redux allow developers to replicate many of the same benefits in JavaScript.

There is an important difference between the way application data is modeled in Elm and in ReasonReact. Elm has a single record, called the "model", which contains the state of the entire application. Actions created inside any component will bubble up to the top level reducer, which will then update the model. ReasonReact, however, stores state at the level of an individual component. If a component wants to modify the state of one of its parents, it must do so via a callback. In my application, I chose to store the entire state in a top-level `Page` component, and pass callbacks to sub-components like `CatalogMove` that need to update that state. However, these callbacks make the code less readable, and they could become increasingly difficult to understand with deeply nested components.

I think that Elm's approach is easier to use out of the box. Storing the entire application state in one place is conceptually simple, and makes it easy to debug code. Elm's command-line development server also provides a user interface to view the current application state as well as its state at any time in the past, and the sequence of actions that have modified it. On several occasions, I was able to fix bugs right away because I saw that an action was not being triggered correctly, or was not modifying the model as expected. That said, the ReasonReact community does maintain a version of Redux, known as Reductive, which provides single-source state management in Reason. While this does require mixing two different ways of storing application state, it also means that Reason developers have more flexibility in how they structure their applications. This may be attractive for some developers, although I personally found Elm's approach to be easier to work with.

I also ran into one particular point of confusion when working with Reason. In order to use action/reducer systems, the application state usually has to be immutable, so that it is easy to detect which parts of the state have been modified and only re-render corresponding parts of the DOM. In Elm, all values are immutable, so the language inherently enforces this requirement, helping to prevent bugs. In Reason, however, certain value are mutable. For example, `Js.Array.removeCountInPlace` is a library function that mutates an existing array by removing some number of elements at a given index. I found that if I use that function inside a reducer to modify a deeply nested part of the application state, the application UI is updated correctly, even though the state is still a reference to the same record as before. As far as I can tell, this might work for two reasons:

(1) ReasonReact happens to be checking the entire state in this case, but it won't always do so as the state gets more complicated. I'm getting lucky right now, but this will lead to bugs in the future.
(2) ReasonReact always checks the entire state before updating the DOM, rather than only looking at parts of the state that appear to have been modified. As the state becomes more complicated, updates will become increasingly slow.

Neither of these is a good thing, and there may be a third explanation that doesn't lead to bugs or slow applications. However, I haven't been able to find one. The strict data immutability in Elm is comforting when writing an application using the action/reducer paradigm, and I wish that Reason had a similar guarantee so that I didn't have to worry about these issues.

3.3 Defining DOM Structure

Both Elm and ReasonReact provide a syntax for defining UI components that maps directly to the DOM. In Elm, each component is a function (e.g. `div` or `span`) that takes two arguments: its attributes and its children. In Reason, components are defined using JSX syntax, similar to React. Here is the same component defined in HTML, Elm, and Reason:

HTML:

```
<div class="input-wrapper tag">
  <input class="input" type="text" />
  <button class="delete-button">X</button>
</div>
```

Elm:

```
div [ class "input-wrapper tag" ]
  [ input [ class "input", type_ "text" ] []
  , button [ class "delete-button" ] [ text "X" ]
  ]
```

Reason:

```
<div className="input-wrapper tag">
  <input className="input" _type="text" />
  <button className="delete-button">
    (ReasonReact.stringToElement("X"))
  </button>
</div>
```

Reason syntax is much easier to read if coming from an HTML background, and even after spending a decent amount of time with Elm, I still found it more difficult to parse visually, especially if the DOM is large and complex. I think this is because in HTML/JSX it is easy to find pairs of opening and closing tags, so the nested structure of the DOM is immediately apparent. In Elm, there is no such thing as a "closing tag", so it is harder to see that structure. However, Elm is also conceptually simpler—you are simply calling functions without introducing any special syntax. The compilation from Reason JSX to OCaml also caused some confusing error messages, which I discuss in the **Error Messages** section.

Another difference arises when breaking down the DOM structure into smaller components. In Elm, you can simply define arbitrary functions that take whatever parameters you like, and return a DOM component. Reason, on the other hand, encourages you to write special modules that inherit methods from the ReasonReact library, and can be written directly into JSX in the same way as default HTML tags, e.g. `<CatalogMove />`. Again, Elm's approach is conceptually simpler, while Reason's approach is a little more readable. However, I found both approaches to be fine in this case.

3.4 Type Systems

Elm and Reason have largely similar type systems. They both ensure that function arguments, return values, and other operators are correctly typed, and raise compile time errors when they are not, preventing a whole class of errors that can readily arise when writing native JavaScript code. I found the strict typing to be very refreshing—rather than slowing me down, it felt like I was able to write code more quickly, since I did not have to worry about double

checking the types of function arguments or the names of record fields after making a change.

Elm and Reason both support records (the equivalent of JavaScript objects), algebraic data types including tuples and sum types (called union types in Elm), type aliases, type inference, and optional type annotations. Unlike Reason/OCaml, Elm does not support labeled function arguments. This was not a problem for me, but in a more complicated application, labeled arguments could certainly be convenient—although passing a record to the function might be an equally good solution. Both languages also provide expressive pattern matching syntax, and a compiler guarantee that all possible values will be matched. My type definition for a swing dance move in both languages is as follows:

Elm:

```
type alias Move =
  { id: Int
  , name: String
  , tags: Array String
  , urls: Array String
  }
```

Reason:

```
type move = {
  id: int,
  name: string,
  tags: array(string),
  urls: array(string)
};
```

While these type definitions are very similar, the type systems of the two languages do differ, especially when dealing with records. In Elm, if you write

```
myFunc r = r.a + r.b
```

the compiler infers that the type of `myFunc` is

```
<function> : { a | a : number, b : number } -> number
```

i.e. it infers that `r` is a record with two number fields, `a` and `b`. However, if you write the equivalent function in Reason:

```
let myFunc = (r) => r.a + r.b;
```

you get a compilation error, `Unbound record field a`. This is because in Reason, records cannot exist unless their structure has an explicit type definition. The following Reason code compiles fine:

```
type recordType = { a: int, b: int };
let myFunc = (r) => r.a + r.b;
```

It isn't necessary to give `myFunc` a type annotation—the compiler will infer that its argument is of type `recordType`. However, that type definition must exist somewhere, or the compiler will complain.

In addition, Elm makes it easy to use nested records, either through type inference or through an explicit type alias:

```
type alias RecordType = { a: { b: Int, c: Int } }
```

In Reason, writing

```
type recordType = { a: { b: int, c: int } };
```

results in an error, `Record type is not allowed` (which, incidentally, is not a very useful error—see the **Error Messages** section for more on error messages in the two languages). However, rewriting the code in the following way eliminates the error message:

```
type subRecordType = { b: int, c: int };
type recordType = { a: subRecordType };
```

In Reason, nested records must have their own separate type definitions.

Overall, Elm's type system has the advantage of allowing developers to progressively build up code with complex record types, without worrying about adding type annotations. This meshes with Elm's philosophy of writing first and adding type annotations later. Reason requires you to plan ahead more if you want to use records, but ultimately forces you to write more readable code and break down your model into digestible parts, which improves code quality. In my case, since I already knew the structure of my model, I did not mind having to explicitly declare all my record types, and Reason worked well for me. Generally speaking, I found both type systems to be very expressive, and I don't think either is clearly better.

I was, however, unhappy to learn that in order to write a simple application in ReasonReact, I had to use code that is not type-safe. Reason does not have a well-typed way to access the new value of a text input inside the `onChange` listener. You must use a function like the following, which is equivalent to accessing `evt.target.value` in JavaScript:

```
let valueFromEvent = (evt) : string => (
  evt
  |> ReactEventRe.Form.target
  |> ReactDOMRe.domElementToObj
)##value;
```

This function finds the DOM node that was the target of the event, and retrieves its `value` property. However, Reason has no way to know that this property is a string, except the type annotation on the `valueFromEvent` function. If `value` is in fact of a different type, the program could crash at runtime. I never had to do something unsafe like this when using Elm, and I suspect that similarly unsafe constructs might arise in other common situations when using Reason (although I did not run into any).

3.5 Language Syntax

Elm has a syntax reminiscent of other functional languages. There are no parentheses around function arguments and no semicolons, and it makes use of `let ... in` statements. Reason, while only a thin layer on top of OCaml, strives to have a syntax much closer to JavaScript. `let` statements are followed by semicolons, function arguments are surrounded by parentheses, and code blocks are surrounded by curly braces. Functions are defined using the "arrow function" syntax introduced in ECMAScript 6. Coming from a JavaScript background, I found Reason's syntax to be more welcoming and readable. I think that especially for teams transitioning from JavaScript, this is a selling point, since it makes the mapping from imperative to functional concepts easier to see.

That said, once I was used to Elm's syntax, I found it to be just as easy to work with (apart from defining the structure of the DOM, where I found JSX to be more readable). Using JavaScript syntax also comes with some trade-offs. For example, functions in Reason that

take multiple arguments look like "normal" JavaScript functions. For example:

```
let myFunc = (a, b) => a + b;
```

However, this function is curried under the hood, and it is perfectly valid to partially apply it:

```
let partiallyAppliedFunc = myFunc(10);
```

For a JavaScript developer, the original function definition is easy to understand, but the partially applied function could be very confusing. Since syntax is new from the beginning in Elm, developers are more likely to realize that function application works differently, and less likely to be confused by function calls.

Another area where both languages are lacking is an immutable update syntax to change a field that is deeply nested within a record. This is a use case that arises frequently when writing programs using the action/reducer paradigm. When writing JavaScript code, Redux provides a `combineReducers` function that makes it easy to update a complicated application state with a deeply nested structure, and do so without mutating any objects. Elm and Reason both have simple syntax to immutably update a record and change fields at the top level, but updates to nested fields become much uglier. Below is the syntax I use to update the application state when the user changes the name of a move. In both cases, `name` refers to the new name entered by the user, and we want to update the state so that `state.draft.name` is equal to this new name.

Elm:

```
let draft = model.draft in
let newDraft = { draft | name = name } in
({ model | draft = newDraft }, Cmd.none)
```

Reason:

```
let draft = { ...(state.draft), name };
ReasonReact.Update({ ...state, draft })
```

Both languages require multiple lines to make this change, and if `name` were nested further, the update would become even more complicated.

3.6 Error Messages

Elm is a clear winner in this area. Elm's error messages were friendlier, easier to understand, and more consistent than Reason's error messages. Both languages make an effort to provide readable error messages. For example, in Elm, writing

```
myVar = if (1 < 2) then "one" else 2
```

leads to the following error message:

```
Detected errors in 1 module.

-- TYPE MISMATCH ---------------------------------------------------- index.elm

The branches of this `if` produce different types of values.

6| myVar = if (1 < 2) then "one" else 2
                ^^^^^^^^^^^^^^^^^^^^^^^

The `then` branch has type:

    String

But the `else` branch is:

    number

Hint: These need to match so that no matter which branch we take, we always get
back the same type of value.
```

In Reason, writing

```
let myVar = if (1 < 2) { "one" } else { 2 };
```

leads to the following error message:

```
We've found a bug for you!
/Users/jackswiggett/Documents/Stanford/4-Senior/1-Fall/CS242/reason/src/page.re 1:39-43

1 | let myVar = if (1 < 2) { "one" } else { 2 };

This has type:
  int
But somewhere wanted:
  string

You can convert a int to a string with string_of_int.
```

Both messages clearly show the section of code that caused the error, explain why the error occurred, and provide hints as to how to resolve it. However, in practice I often ran into much less friendly error messages when using Reason. For example, if I had forgotten the parentheses in the `if` statement and instead written

```
let myVar = if 1 < 2 { "one" } else { 2 };
```

I would have gotten the following error

```
FAILED: src/page.mlast
/Users/jackswiggett/.nvm/versions/node/v8.9.1/lib/node_modules/bs-platform/lib/bs
c.exe -pp "/Users/jackswiggett/.nvm/versions/node/v8.9.1/lib/node_modules/bs-plat
form/lib/refmt3.exe --print binary" -ppx '/Users/jackswiggett/.nvm/versions/node/
v8.9.1/lib/node_modules/bs-platform/lib/reactjs_jsx_ppx_2.exe'   -w -30-40+6+7+27
+32..39+44+45+101 -bs-suffix -nostdlib -I '/Users/jackswiggett/Documents/Stanford
/4-Senior/1-Fall/CS242/reason/node_modules/bs-platform/lib/ocaml' -no-alias-deps
-color always -c -o src/page.mlast -bs-syntax-only -bs-binary-ast -impl /Users/ja
ckswiggett/Documents/Stanford/4-Senior/1-Fall/CS242/reason/src/page.re
File "/Users/jackswiggett/Documents/Stanford/4-Senior/1-Fall/CS242/reason/src/pag
e.re", line 1, characters 16-17:
Error: 489: <UNKNOWN SYNTAX ERROR>
File "/Users/jackswiggett/Documents/Stanford/4-Senior/1-Fall/CS242/reason/src/pag
e.re", line 1, characters 0-0:
Error: Error while running external preprocessor
Command line: /Users/jackswiggett/.nvm/versions/node/v8.9.1/lib/node_modules/bs-p
latform/lib/refmt3.exe --print binary '/Users/jackswiggett/Documents/Stanford/4-S
enior/1-Fall/CS242/reason/src/page.re' > /var/folders/3d/rgspr8615tdcwy7k1b6q7hd0
0000gn/T/ocamlpp38e33a
```

This error is cluttered and difficult to process. The bold text does not even show the correct location of the syntax error—it is written in unbolded text above. And the error message is `<UNKNOWN SYNTAX ERROR>`. In practice, I frequently saw unknown syntax errors while working in Reason, whereas I never saw an error message from Elm that wasn't easy to process and act on.

Reason also caused some cryptic errors due to the process of converting from JSX to OCaml. For example, I ran into the following error during development:

```
We've found a bug for you!
/Users/jackswiggett/Documents/Stanford/4-Senior/1-Fall/CS242/reason/src/page.re 240:13-39

238 |  <div className="flex-line">
239 |    <span className="input-label">
240 |      ReasonReact.stringToElement("Name:")
241 |    </span>
242 |    <div className="input-wrapper">

You're missing arguments: string
```

This error occurred because in JSX, `ReasonReact.stringToElement` and `("Name:")` are interpreted as two different children of the span, rather than as a function call. This is immediately clear from the compiled OCaml code:

```
((span
  ~className:"input-label"
  ~children:[ReasonReact.stringToElement; "Name:"]
  ())[@JSX ])
```

We can see that `ReasonReact.stringToElement` is missing the `string` argument. Wrapping the function call in parentheses solves the problem:

```
<span className="input-label">
  (ReasonReact.stringToElement("Name:"))
</span>
```

In this case, the Reason compiler eagerly exclaims "we've found a bug for you", but the error message provides no useful insight into how to fix the bug. While Elm's error messages felt friendly, I grew to hate Reason's overzealous "we've found a bug" message.

3.7 Libraries and Documentation

Both languages provide fairly robust standard libraries. One confusing aspect of Reason is that its primary standard library is a one-to-one port of the OCaml standard library—but it also provides direct interfaces to the JavaScript standard library. For example, you can retrieve the length of an array with `Array.length`, or with `Js.Array.length`. Certain functions only exist in one library or the other, and it was sometimes confusing to have to look in several different places. Elm provides a single set of core libraries that are easier to understand and navigate. While the `Js.*` libraries feel tacked on to Reason, all of Elm's libraries feel native.

Both languages provide fairly detailed documentation of their standard libraries. In Reason, developers can also look at other OCaml references and translate the syntax from OCaml to Reason. That said, the Elm documentation is generally easier to read and provides clearer examples. In addition, the `Js.*` libraries in Reason provide almost no documentation except the type signature of each function. This can be very confusing, since many JavaScript standard library functions take a variable number of arguments, something which is not possible in Reason, and these have often been split into multiple different functions with different names in order to allow for inclusion or omission of certain arguments.

3.8 JavaScript Interoperability

I did not have to interface directly with JavaScript when developing this application. However, this is obviously a priority for teams that want to gradually add functional components to an application currently written in JavaScript, or need to use JavaScript libraries from within Elm or Reason.

3.8.1 Elm. A piece of user interface written in Elm can be easily embedded in some larger program by simply loading that Elm program into a given DOM element. For example, you could load a `Main` module from Elm into `<div id="main"></div>` with the following code. The `main.js` file is the compiled code generated by Elm.

```
<script src="main.js"></script>
<script>
    var node = document.getElementById('main');
    var app = Elm.Main.embed(node);
</script>
```

Elm provides two ways to communicate with JavaScript. First, "flags" can be passed to the `Elm.Main.embed` function. These are arbitrary key/value pairs that can be used to configure the Elm module when it is initialized. Second, Elm provides an API to send and receive messages to/from JavaScript, through what are known as "ports". Thus Elm code can request that a JavaScript library process some data or perform an action, and receive a message once that action is complete. Data sent through ports is strictly typed, and allows for a wide variety of possible data types. Elm validates the type signature of any data sent from the JavaScript side, so that a runtime error will be thrown in JavaScript if a port is used incorrectly, and invalid data will never make it to the Elm part of the application.

3.8.2 Reason. Reason also makes it easy to embed components in a larger JavaScript application. As with Elm,

```
ReactDOMRe.renderToElementWithId
```

can be used to load a Reason component inside an arbitrary `div`. In addition, ReasonReact provides helper functions, `wrapJsForReason` and `wrapReasonForJs`, that make it easy to use ReasonReact components inside ReactJS and vice versa.

While Elm abstracts away interaction with JavaScript code and emphasizes type safety, Reason makes it easy to add arbitrary (and unsafe) JavaScript in the middle of a program:

```
Js.log("This is Reason");

[%%bs.raw {|
console.log('This is raw JavaScript');
|}];
```

Raw blocks of JavaScript can return values back to Reason. By default, any values returned back to Reason have a special type that will unify with any other type, meaning that they can be used right away without any type annotation, but all the benefits of Reason type checking are lost. The developer can provide a type annotation, but if that annotation is incorrect, the program could of course crash at runtime.

Reason provides some other slightly more elegant methods for calling JavaScript functions and executing code, but they all involve directly using JavaScript values within Reason, rather than using a type-safe messaging interface like Elm does. This approach certainly makes it easier to quickly include JavaScript code and prototype an application. However, I imagine that it would lead to more errors, and more confusing errors, in the long run.

4 RESULTS: QUANTITATIVE METRICS

I wrote roughly the same amount of code in both languages. Excluding comments and blank lines, I have 232 lines of Elm code and 262 lines of Reason code, which I do not consider to be a significant difference.

I saw a much larger difference in the size of the compiled JavaScript output. The output from Reason was 976 KB, whereas the output from Elm was only 218 KB. This is almost certainly primarily due to the libraries included in the compiled output. It's possible that the Reason output includes JavaScript ports of many OCaml library functions, whereas the Elm API is optimized to make use of native JavaScript libraries where possible, which could contribute to the size disparity. However, that's only a conjecture.

After minifying the compiled JavaScript, the output from Reason had a total size of 296 KB, whereas the output from Elm had a total size of 76 KB.

I did not notice any appreciable difference in the performance of the two web applications. Both ran very quickly. A more performance-intensive use case would be needed in order to see whether one of the two languages leads to better rendering speed or performance.

5 CONCLUSION

Both Elm and Reason provide many advantages over writing JavaScript web applications. They make it easier to iterate without worrying about introducing subtle bugs, and they help enforce good code quality. They both enforce an action/reducer system for dealing with application state, which makes it easy to pin down bugs and keep the application well-organized. While they have comparable type systems and language features, Elm generally has much better error messages than Reason. It also has a simpler standard library, and higher-quality documentation. Reason, however, has a syntax that is much friendlier to developers used to working with JavaScript and HTML. Ultimately, choosing one framework over the other will depend on the project at hand and the past experience of team members. Many of the benefits gained by writing in Elm and Reason can also be achieved by using a library such as Redux, and by being strict about adding type annotations with Flow or TypeScript. However, especially for new projects where developers are willing to try something different, I think that Elm and Reason are both very good choices for web application development.

6 REFERENCES

The Elm website has tutorials, examples, and documentation for the Elm programming language: http://elm-lang.org/.

The Reason website has tutorials, examples, and documentation for the Reason programming language: https://reasonml.github.io/.

Information about ReasonReact is available at https://reasonml.github.io/reason-react/.

I also made use of a ReasonReact tutorial by Jared Forsyth, available at https://jaredforsyth.com/2017/07/05/a-reason-react-tutorial/.

Information about Redux, a JavaScript library for managing application state that works similarly to Elm and ReasonReact, is available at https://redux.js.org/.