OULU UNIVERSITY OF
APPLIED SCIENCES

Matthias Kern

LEARNING REASONML BY EXAMPLE

LEARNING REASONML BY EXAMPLE

Matthias Kern
Bachelor’s Thesis
Spring 2018
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author: Matthias Kern
Title of the bachelor’s thesis: Learning ReasonML by example
Supervisor: Janne Kumpuoja
Term and year of completion: 2018   Number of pages: 29 + 12

ReasonML is a new dialect of the OCaml language targeted at modern web development. The aim of this bachelor’s thesis was to explore the promises and core paradigms of the language. The aim was also to evaluate its approachability and usability.

The background of web development and functional programming in relation to ReasonML are explored. The syntax and features of ReasonML are elaborated. For the purpose of studying and evaluating the language, an interactive game was created. The game is an implementation of Conway’s Game of Life.

The demo project was created successfully and without major issues. ReasonML’s static type system and compiler messages were evaluated as useful. The syntax of the language was determined to be simple to learn for JavaScript developers. This was considered was considered to be suitable for building these types of applications.

Keywords:

JavaScript, ReasonML, OCaml, Client-Side, Web Development, Front-End

PREFACE

The finished project can be viewed online. It is best viewed on a desktop computer with a modern web browser, such as Google Chrome or Mozilla Firefox.

The source code repository for the demo project is also available openly online on the website GitHub and it can be run and followed along while reading.

There are only a few resources of the topic of ReasonML and they are all online. The first book of the subject, Dr. Axel Rauschmayer's "Exploring ReasonML and functional programming" is still in progress and can be read online. A lot of knowledge in this work also originates from the official documentation.

The author would like to thank the thesis supervisor Janne Kumpuoja, the ReasonML contributors, and the community. Special thanks to the organizers of Reason-Conf 2018.

Helsinki, 21.05.2018

Matthias Kern

CONTENTS

VOCABULARY

API:	Application programming interface
bs:	BuckleScript package prefix
CSS:	Cascading stylesheets
DOM:	Document object model
GCD:	Greatest common divisor
HTML:	Hypertext markup language
JS:	JavaScript
JSX:	JavaScript XML
NPM:	Node package manager, a package manager for JavaScript
WebGL:	Web Graphics API, based on OpenGL
XML:	Extensible Markup Language

1 INTRODUCTION

ReasonML is a relatively new programming language that promises to bring the features and ecosystem of the systems language OCaml together with the primarily web-focused language JavaScript. The author has worked with and studied the programming language JavaScript extensively. He also has experience with the TypeScript language. The focus of this thesis is to explore the language ReasonML and evaluate it from the author's context. Starting off, the background and context of ReasonML in web development and programming languages are discussed in chapter two. In the subsequent chapter, the characteristics of the language and its relation to other language are illustrated. For the purpose of evaluation, an interactive web-based game has been created with ReasonML. The demo project has been chosen to help with the evaluation and has had a set of predefined goals that are elaborated in chapter four. The final chapter discusses the author's conclusions and provides a view into the future of ReasonML.

The topic of the thesis has been chosen out of personal interest in functional programming and programming languages in general. The author hopes that the exploration of the topic will lead to a better understanding of programming languages and their characteristics. He also wishes this thesis to be an introduction to the OCaml language for himself. The techniques and insights learned while working on this thesis are expected to be of help later in the author's career as a software engineer.

2 BACKGROUND

This chapter is meant to give an overview of the context and work of this thesis. It should provide an insight into the background of programming web applications and functional programming with a focus on JavaScript.

2.1 Web Development

Web development covers all aspects of work that is performed to produce websites. It is a wide field that is spanning all kinds of industries and it ranges from simple static pages to more complex applications that come very close to native desktop applications. As more complex and resource-intensive applications have been developed to work in the browser and code bases have been growing as well, the need for performant but at the same time manageable code has been increasing. For client-side web development, also called “front end” the traditional stack has included HTML to create site markup, CSS for styling, and JavaScript for interaction scripting. The development of new technologies in the field was traditionally limited to the fixed constraints provided by the browsers and the need of keeping things backward-compatible. To be able to use advanced or even unreleased features before they are a standard in commonly used browsers, engineers have been working on compilers. These compilers can translate, for example, the superset SASS to compatible CSS.

For traditional server-rendered websites, developers were able to use whatever stack of technology they preferred in the backend. In recent years the method of creating so-called client-side web applications in JavaScript has gained a lot of popularity. JavaScript is an untyped language and does not provide run-time type safety by the browser interpreter. This fact has resulted in engineers working on ways to introduce static type checking into the language. One example is the Google Closure Compiler, which derives types from annotations provided in the JSDoc format. (1) Later on, supersets of JavaScript, which include type declarations, were created, such as Typescript or Flow. The strictness of the type checking of these tools depends on the configuration of the respective tool and on how strongly the user uses types inside the source code. Languages that enforce even stronger type checking, but are higher level abstractions, are for example PureScript, ClojureScript or Elm.

2.2 Functional Programming

Different programming languages have adopted functional programming in different ways. The core concepts of the paradigm though are pure functions that behave like mathematical equations and immutable data structures. The term “pure” describes functions that will yield the same result with the same input every time they are invoked. They will also not change state outside of the function scope or produce so-called “side-effects”. This allows for a better testability and readability, as every pure function can, in theory, be tested and read without the context. As such, the functions can also be composed easily, which is another commonly used technique in the functional programming paradigm. The opposite of immutable is mutable, which means being able to change or mutate a value directly. Instead, a clone of the value is made, and the changes are applied. Languages that support functional programming include Scheme, Erlang, Haskell, OCaml and ReasonML.

JavaScript was designed to have a C-like syntax as Java, and for a long time, it was common to use it in a similar approach with object-oriented programming techniques. Since the ECMAScript version 5, new language features have been added to the core language specification to enable functional programming such as map, filter or reduce functions on the Array prototype object. (2) In 2015, the Version 6 introduced arrow functions and other features to improve the immutable use of objects. (3) Libraries, e.g. Ramda or Lodash have been used to help with processing data or writing algorithms in a functional style. Modern client-side libraries, such as Angular or React, have been able to showcase the performance benefits of using immutable data and pure functions. Because of these developments and the urge of developers to write readable, well-functioning and maintainable code, functional programming has found its way to a large audience of web developers

3 REASONML

In the following chapter, the history and characteristics of ReasonML are described. Subsequently, the language is depicted in the context of functional programming.

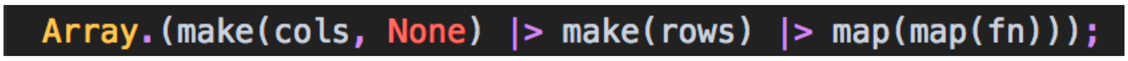
3.1 Basic Information and History

ReasonML is not, in fact, an entirely new language, but rather a syntax on top of the language OCaml. OCaml is an "industrial strength programming language" and supports according to the official website “functional, imperative, and object-oriented styles”. (4) OCaml has powerful language features with a static type system that is also able to do type inference. OCaml can be associated to the family of ML-style languages. These so-called "Meta-languages" are useful for handling and processing of other languages and in general tree-type structures. They are also used in writing compilers or static analyzers. However, they are not known for wide-spread use in web development.

Jordan Walke, the inventor or React, arguably one of the most popular client-side libraries for interacting with the DOM at the moment of writing, prototyped it early on in SML, a relatively close language to OCaml. At Facebook, the company where Reason was created, OCaml had been in use for several years, and e.g. the Flow static type checker had been written in OCaml. (5) After React became popular, Jordan Walke and Cheng Lou returned to the original implementation and began iterating on Reason and its syntax with the goal of creating a bridge between JavaScript and OCaml. (6) A strong interoperability and a familiar, friendlier syntax, which make the OCaml advantages and ecosystem accessible for JavaScript developers, are named as some of the main goals of the project. (7). ReasonML, according to Jordan Walke, “addresses the biggest problems that I’ve observed in building UI applications over the last five years”, “In short, it is the best way to take React to the next level”. (8)

3.2 Modules

In Reason, every file is a module. BuckleScript, the Reason compiler, chooses up all Reason files in the target directory by default. All *.re* files in the directory are immediately accessible from other code under their Module namespace. Modules can also be created nested inside other modules with the Module keyword. By default, all fields of a module are exposed, but there is an option to explicitly define their public API with *.rei* signature files. To ease the usage of modules, they can be "opened" via an open keyword which makes its functions available in the current block statement. They can also be opened inline in an expression to make source code more legible and concise. Figure 1 shows an example of the inline usage of the make and map functions from the Array module.



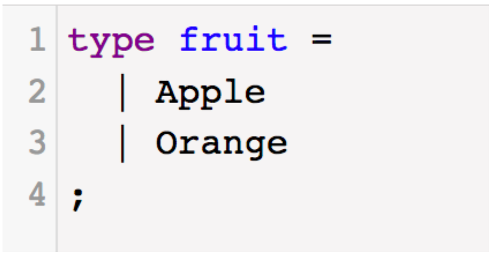
```
Array.(make(cols, None) |> make(rows) |> map(map(fn)));
```

FIGURE 1. Inline module opening example

3.3 Variant types

JavaScript is a dynamically typed language that allows implicit coercion, the conversion of types from one to another, which has led to a lot of grudges overtime against the language. Variables can also hold the value of null or undefined which has been a source of countless bugs in any language implementing this concept. Tony Hoarse, the inventor of the null reference, called the concept "historically a bad idea" (9).

Reason promises strong type inference and expressiveness with a powerful feature called variants. Variant types share a similarity with the Enum type in C-style languages. Variants consist of type constructors. These symbols can contain arguments and can as such be used to semantically display dynamic values and state as is visible in Figure 2, where the type fruit is either of the type Apple or Orange.



```
type fruit =
  | Apple
  | Orange
;
```

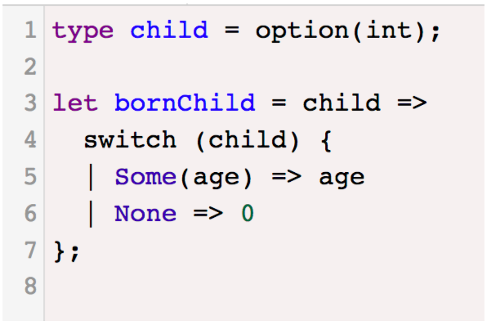
FIGURE 2. Example usage of a variant type

If one wants to extend or reuse them, polymorphic variants exist, which\ make using the same type constructor in multiple variants possible (10). Polymorphic variants also enable the use of generic and variable types.

3.4 Pattern Matching

Variants enable another powerful feature called Pattern Matching. Reason is not the first language to feature this concept, and there is even a proposal for an addition to the JavaScript standard being discussed. (11)

Another example of a variant is the option type thatt is built into the language. It forces the distinction between not initialized or “empty” and initialized values with the help of the Some(argument) and None type constructors. Figure 3 shows an example usage of the type with pattern matching.



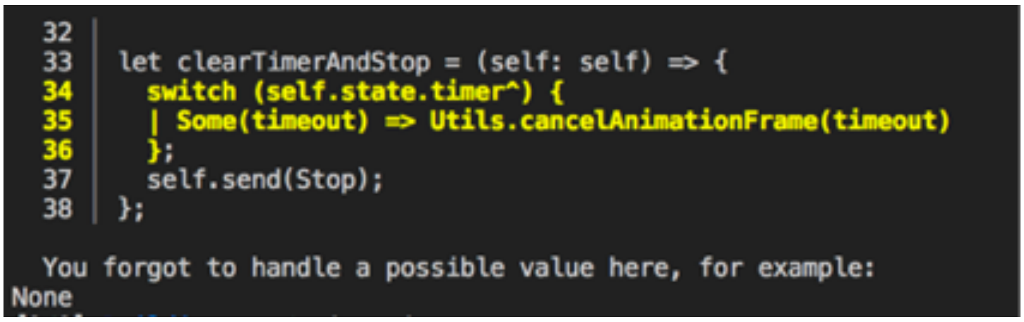
```
type child = option(int);

let bornChild = child =>
  switch (child) {
  | Some(age) => age
  | None => 0
};

```

FIGURE 3. Example usage of the option variant type

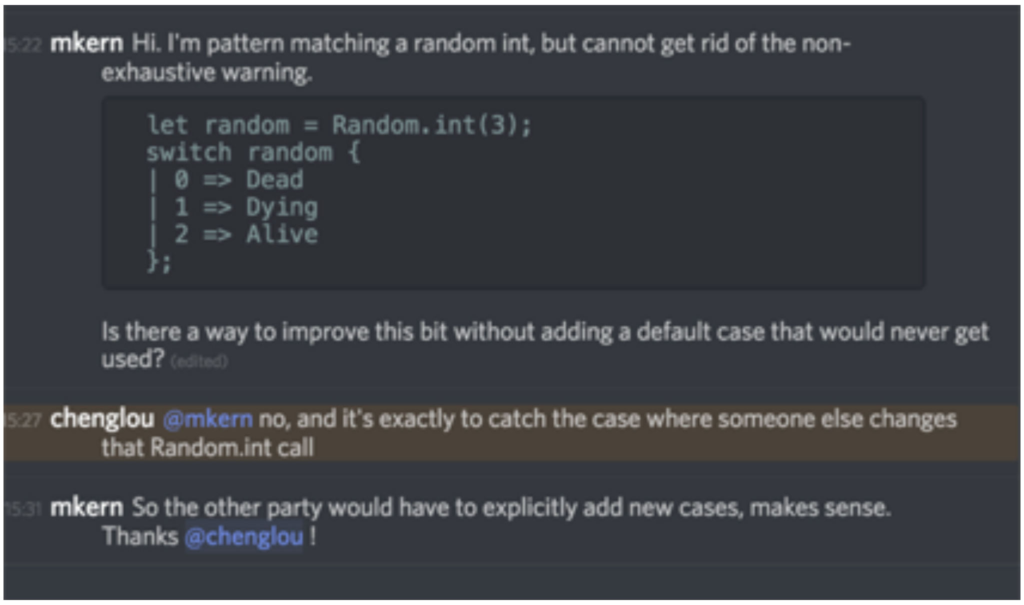
On the first glance, the switch statement looks familiar and seems to work similarly like the one found in JavaScript. However, the Reason equivalent does not only work with primitive values, but also with type constructors and complex bindings, such as tuples, lists or even exceptions. There are also so-called type guards thattt allow for even more complex conditions inside the branches. Combined with the ability to bind the returned value from the statement, pattern matching removes the need for a frequent pattern in C-Style languages: several if/else clauses that follow each other. The if/else pattern has revealed itself to be a common source of bugs where values would lead to a wrong branch due to misbehaving conditions or because they were forgotten edge cases. The ReasonML compiler warns the developer when a switch statement does not match all cases to prevent uncertain error states, even if that goes along with writing more code (Figure 4). One of the core developers explains this behavior as being an explicit design decision, to prevent errors in advance (Figure 5).



```
let clearTimerAndStop = (self: self) => {
  switch (self.state.timer^) {
  | Some(timeout) => Utils.cancelAnimationFrame(timeout)
  };
  self.send(Stop);
};

You forgot to handle a possible value here, for example:
None
```

FIGURE 4. Compiler warning for non-exhaustive pattern matching of an option



5:22 **mkern** Hi. I'm pattern matching a random int, but cannot get rid of the non-exhaustive warning.

```
let random = Random.int(3);
switch random {
| 0 => Dead
| 1 => Dying
| 2 => Alive
};
```

Is there a way to improve this bit without adding a default case that would never get used? (edited)

5:27 **chenglou** @mkern no, and it's exactly to catch the case where someone else changes that Random.int call

5:31 **mkern** So the other party would have to explicitly add new cases, makes sense. Thanks @chenglou !

FIGURE 5. Screenshot of chat conversation regarding non-exhaustive pattern matching from Cheng Lou (Facebook)

3.5 Interaction with JS and OCaml

Listed as other aims of the Reason project are compatibility and interoperability with the existing code. BuckleScript utilizes OCaml's macro programming abilities to handle these foreign function interfaces with specific decorators inside the code. (12) With the help of these macros, the existing modules from NPM and interfaces implemented by the browser can be used with relative type safety within the Reason code. The point of pressure then lies on these bindings to be correct, which requires on one hand knowledge of the code that is wrapped but on the other hand forces one to think carefully about typing. The OCaml code can be used in the same way as Reason modules but it needs to be used carefully as some modules may contain native modules which will not work inside the browser.

3.6 Differences to OCaml

The Reason syntax has been iterated heavily and it is still progressing. The current version used in this work is 3.1.0.

In order to appeal and make the transition for JS developers easier, the engineers working on Reason have removed some of OCaml's syntax characteristics which are especially divergent and adjusted others to be closer to JavaScript's C-style syntax. Normal functions are written using => lambda notation, a syntax that has been added to the recent version of JavaScript and is also often referred to as an "arrow" function. Contrarily, OCaml has several different ways of specifying the same functions, which may not be intuitive to use for users with little experience of ML-style languages.

Reason differentiates between two types of equality: structural, meaning that the content of the compared bindings is identical and referential, meaning that they refer to the same binding. Table 1 shows that the corresponding operators have been adjusted to use an equal number of signs as in JavaScript even if they do not work in an exactly similar way (JavaScript differentiates instead between loose and strict equality).

For the structural inequality comparison the syntax has been adjusted to use the exclamation mark, commonly the Boolean *not,* in the C-Style syntax, OCaml instead uses != to dereference bindings from their original (Table 2).

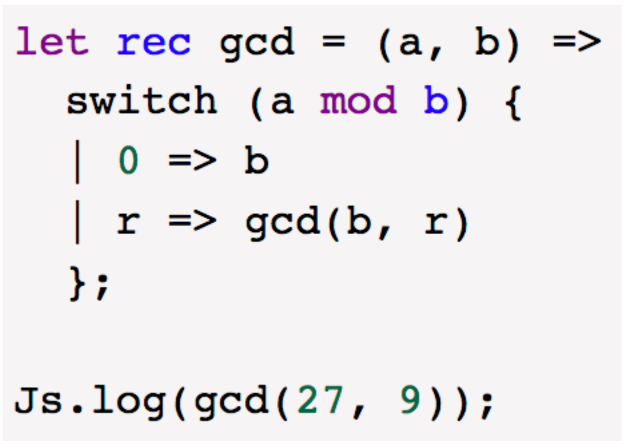
TABLE 1. Comparison of equality operators in OCaml and ReasonML (13)

Equality	**OCaml**	**Reason**
Structural	x = y	x == y
Referential	x == y	x === y

TABLE 2. Comparison of inequality operators in OCaml and ReasonML (13)

Inequality	**OCaml**	**Reason**
Structural	x <> y	x != y
Referential	x != y	x !== y

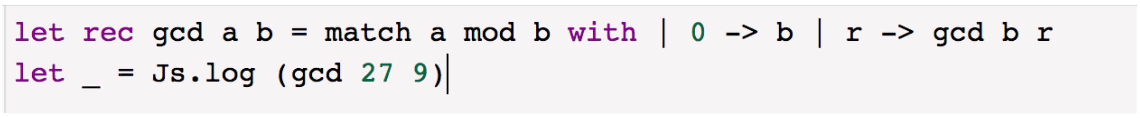
In Figure 6 an example implementation of the greatest common divisor algorithm with Reason is visible along with its OCaml equivalent in Figure 7. This example was taken from the ReasonML Playground. (14) It is apparent that pattern matching has been adjusted to look more like JavaScript's switch statement. Parentheses have been added around function arguments which also looks like the JS equivalent. The imperative logging expression is bound to a return value in reason. ReasonML instead allows to ignore it. This adjustment seems especially useful for using imperative for or while loops. Ocaml also does not use the => operator for the function declaration and uses single arrows -> inside match cases.



```
let rec gcd = (a, b) =>
  switch (a mod b) {
  | 0 => b
  | r => gcd(b, r)
  };

Js.log(gcd(27, 9));
```

FIGURE 6. GCD calculation in ReasonML



```
let rec gcd a b = match a mod b with | 0 -> b | r -> gcd b r
let _ = Js.log (gcd 27 9)
```

FIGURE 7. GCD calculation in OCaml

3.7 Comparison to a purely functional language

OCaml, and as such ReasonML, is not purely functional. Its main focus is on enabling expressive and declarative code and it allows the developer to mix and match between different programming styles and techniques. The language has imperative statements, e.g. for loops, but also alternatives to achieve the same result using the functional paradigm, e.g. recursive functions or higher order functions. This stands in opposition to e.g. Haskell, a purely functional language. Haskell is also statically typed. It has type inference, but it has a totally different way of doing allowing imperative programming and side effects. For example, for or while loops do not exist in Haskell at all.

4 DEMO

The aim of the demo project was to get an introduction to the language and to evaluate it. For this purpose, an interactive website was created where a version of a game of cellular automata is embedded. The concept of the Game of Life was published in 1970 by John Conway. (15) A cell has a status of either alive or dead. The cells can either live or die depending on the status of its neighbors. Depending on their initial position, the cells might form patterns that evolve throughout the game. The game has been chosen because of its need for a number of important concepts that can help evaluate ReasonML. These needs include e.g. iteration over data, generating of random data, state management or recursive functions. As implementation aims of the demo project the following user stories were taken from an online challenge (16):

- “When I first arrive at the game, it will randomly generate a board and start playing.”
- “I can start and stop the board.”
- “I can set up the board.”
- “I can clear the board.”
- “When I press start, the game will play out.”
- “Each time the board changes, I can see how many generations have gone by.”
- “The game should follow the rules of the Game of Life.”

This project is licensed under MIT and it can be found in Appendix 4.

4.1 Development process

The project uses BuckleScript via the bs-platform NPM package to compile the Reason code to JavaScript. Webpack is used to bundle the resulting files into a single one that is referenced from an HTML index page. This file, together with a CSS file containing styles, can be found in Appendix 2. BuckleScript and Webpack are configured and executed with NPM scripts from a package.json file. The configurations for these tools are shown in Appendix 3.

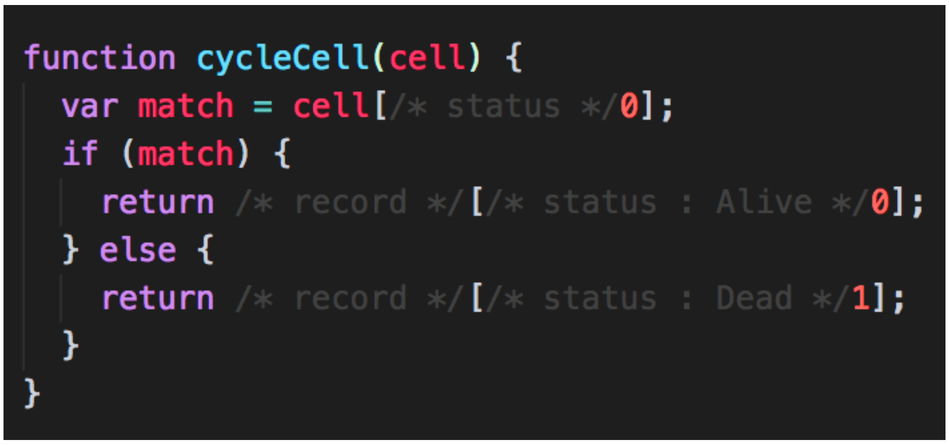
The Reason development tools can be installed via the reason-cli package which provides the refmt code formatting tool as well as the editor completion provider Merlin. BuckleScript and Merlin can both provide type annotations, warnings, and errors in the compiler output and to the used text-editor.

As Reason is in a direct relation to React, it also has first-hand React support. As such, the author decided to use the ReasonReact module. React uses an abstraction called JSX which has an XML-like syntax to describe components and DOM elements. ReasonReact has a couple of differences to the JS equivalent in the way it treats JSX, this is, however, not important for this thesis. The Reason source code, but not the generated JavaScript, can be found in Appendix 1.

4.2 Types

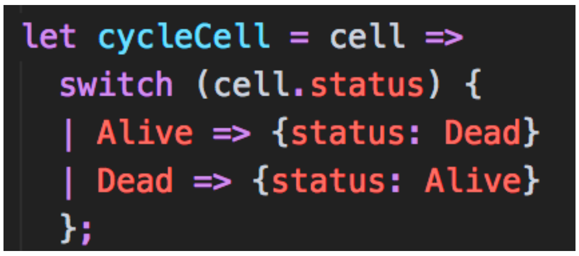
The author generally chose to annotate most functions with explicit types, because he was not used to the strong inference capabilities. The compiler would render probably most of those annotations unnecessary. Annotation can help to enforce and make types explicit. The usefulness of static types revealed itself when refactoring or moving around pieces of code. The author could make sure that other, untouched parts of the application were still intact while the compiler was not emitting error messages.

When inspecting the files BuckleScript generates, it becomes apparent that the compiler is compiling Reason specific types to JavaScript primitives. In Figure 8 the output code for a method cycleCell can be seen annotated with comments that show the original type names. Figure 9 shows the original Reason implementation. The pattern matching statement was compiled to a normal if statement. The cell record was compiled to an Array and the status type to a number. It is possible to compile a ReasonML record to an object in JavaScript, but as no other party relies on the shape of the code, it does not matter what shape the compiler outputs in this case. Even though the structure was modified quite a lot, the code is still readable due to good formatting, preserved naming and added comments.



```
function cycleCell(cell) {
  var match = cell[/* status */0];
  if (match) {
    return /* record */[/* status : Alive */0];
  } else {
    return /* record */[/* status : Dead */1];
  }
}
```

FIGURE 8. BuckleScript output of the cycleCell function



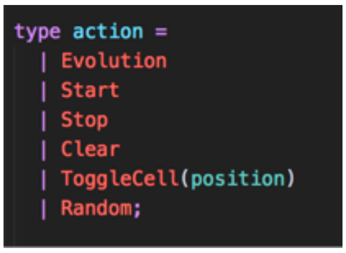
```
let cycleCell = cell =>
  switch (cell.status) {
  | Alive => {status: Dead}
  | Dead => {status: Alive}
  };
```

FIGURE 9. Reason code to toggle the status of a cell

4.3 State Management and React

React uses so-called components as units of modeling in the user interface. In Reason every component should be in its own module and in this project, every component is in a file with the same name.

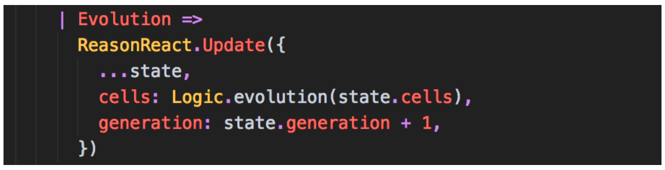
In the demo app, all application state is handled in the App component that is passing down a state record and provides a send function to trigger state changes via actions. This component is a so-called reducer component that passes an initial state and features a reducer method with state and action arguments, which handle the updates of the state. This pattern has become popular in the React ecosystem along with the state management library Redux. (17) In ReasonReact though, this pattern has first-hand support through the ReasonReact.reducerComponent API without the need for an external library. All state changes are defined in a variant type called action. For the demo project, this includes all the user interactions through controls (Figure 10).



```
type action =
  | Evolution
  | Start
  | Stop
  | Clear
  | ToggleCell(position)
  | Random;
```

FIGURE 10. Screenshot of the action variant type

Updates to the state happen inside the reducer function with calls to ReasonReact.Update. Figure 11 shows the example handling of the Evolution action. It seemed odd to the author at first that the return type of this function does not have the return type of state, as it returns a new state, but instead a variant of ReasonReact.update. When looking at the different cases, it becomes apparent that it is used for differentiating between whether a function does trigger an update, no update, or side effects.



```
| Evolution =>
  ReasonReact.Update({
    ...state,
    cells: Logic.evolution(state.cells),
    generation: state.generation + 1,
  })
```

FIGURE 11. Screenshot of a case inside the reducer function

The ReasonReact API is declarative and accessible for someone who has used the React library in JavaScript previously. The reducer component model seems useful, as it abolishes the need for an external state management library and OCaml's combination of pattern matching and variant types seems like a great combination for this purpose.

4.4 Functional programming patterns

The demo program is using immutable data structures, apart from one usage of a mutable let binding. The ref keyword is used to save the id returned from requestAnimationFrame to the state to be able to cancel it later on. In this case, it seemed easier to opt-out of immutable state because it is a side-effect and cannot really be handled in a purely functional way. Especially, for generating the next state of the game, which can be seen in Logic.re, the language seemed to be of advantage. The game board was iterated over cell by cell to generate the next evolution of cells depending on how the status of their immediate neighbors. The author had some issues with writing the functions that map over the cells at first, because he had written the Code in the same style as previously in JavaScript. Being able to utilize of the pipe operator |> made function composition easier reach an understandable and readable form. The function mapCells is one example of that. The author would also probably not have used as much function composition in JavaScript and would have created more intermediate variables. One example is the earlier shown Figure 1, in which the functions of the Array module are used to create a matrix of cells and apply a function fn to the result.

4.5 Result

In the end, all implementation aims were achieved. Figure 12 shows a screenshot of the running demo. The controls "Random", "Start", "Clear", as well as the current generation, are visible on top of the game board. In the top right corner, the current generation of the game state is displayed.

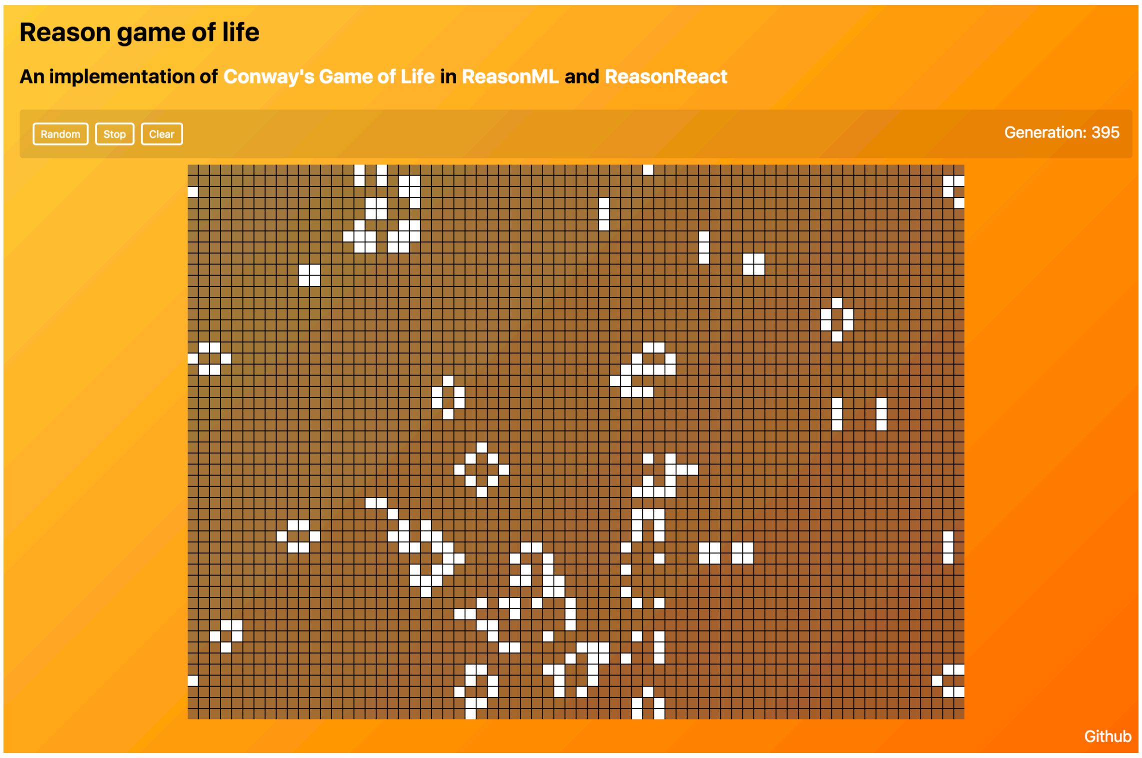
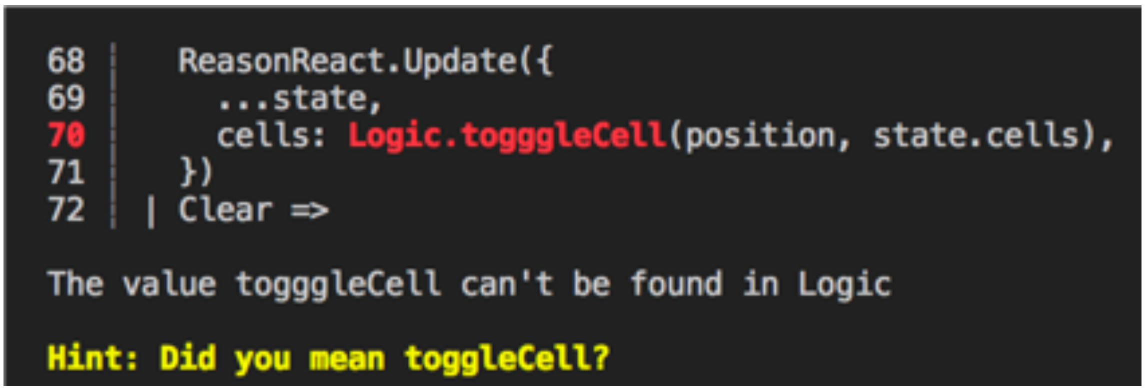

Reason game of life
An implementation of Conway's Game of Life in ReasonML and ReasonReact
Random
Stop
Clear
Generation: 395
Github

FIGURE 12. Screenshot of the running demo project

4.6 Remarks

It took a while for the author to become familiar with some details of the syntax, i.e. when to use a comma operator. The creators' effort to make the transition for developers with a JavaScript-background seems to have succeeded. The author is especially content with how the compiler made refactoring easier with helpful error messages and how simple the initial project setup was. A fast compilation time and a short feedback-loop helped to achieve a good development experience. Little time was needed to spend to search for the error and error messages were generally useful, even hinting at a possible solution. (Figure 13).



```
  ReasonReact.Update({
    ...state,
    cells: Logic.togggleCell(position, state.cells),
  })
| Clear =>

The value togggleCell can't be found in Logic

Hint: Did you mean toggleCell?
```

FIGURE 13. Screenshot of an error message of a not found value

During the process of this work the version 0.4.0 of the ReasonReact library was released. The upgrade to the new version was seamless and without manual refactoring work, as there was an upgrade script available for migrating from the old to the new API. (18)

4.7 Further steps

For a better rendering performance, the program could utilize an external library called reprocessing which renders to WebGL or OpenGL instead of the DOM. With the help of the bsb-native project, it would be possible to cross-compile and embed in native and mobile apps. Some functions could be refactored to be more performant and to utilize of the built-in API better. They could also make use of the new BuckleScript standard library Belt. To improve security while refactoring, tests could be implemented i.e. with the bs-jest library.

5 CONCLUSION AND VIEW INTO THE FUTURE

While implementing the game, the author learned about some of the strengths and weaknesses of the ReasonML language and the ecosystem. It was simple to set up the demo project and after a small initial period, the syntax became familiar. The syntax of OCaml seems unapproachable from a beginner's point of view, and ReasonML is trying to achieve the opposite. The developers' focus on making the transition from JavaScript as easy as possible is obvious.

Reaching into the functional programming paradigm was not always trivial, but Reason's pragmatic approach and well-designed syntax were of advantage. The strong type system was, in general, more empowering than restricting. Semantic variants and pattern matching along with type annotations made the program easy to reason about.

Fast and helpful compiler and editor feedback also resulted in a better developer experience than the author has had previously experienced in a web application project. Remarkable was also the built-in formatting tool, which apparently has also been used to do automatic syntax transforms for Reason upgrades e.g. from the version 2 to the version 3. The documentation was approachable and helpful and especially the official chat channel was a great resource to ask and learn. Only when going into depths of, especially the BuckleScript API, things became confusing, but that was not necessary for the sake of this project. This project also did not have to interact with OCaml native modules.

Looking into the future, Reason is getting better asynchronous code support compatible with JavaScript promises and a new standard library, called Belt. In-browser debugging is also going to be improved through a dedicated debugging mode in Bucklescript.

The meta-programming aspects of OCaml and Reason seem very useful, even if the demo project did not directly utitlize of those features. There is the possibility for user-defined syntax extensions or first-class code generation through a feature called PPX. (19) Those are essentially placeholders that can be over-written by external tools. There are already libraries that utilize this feature, for example for embedding of the GraphQL data query language. (20)

On the horizon for OCaml there are modular implicits, which could allow the use of ad-hoc polymorphism. (21) This would allow for e.g. using the same operator for integer as well as float math operations and not, as might be initially confusing for newcomers, having to use e.g. + and +. or / and /. . (22) On the horizon for OCaml there is also support for concurrency and parallelism through multi-core usage. (23)

The author was able to observe critique from the real-world users of Reason, who often mentioned that the BuckleScript API is difficult to grasp and that bindings to bigger libraries can be difficult to write correctly. Another comment concerned the Reason JS Promise API.

In the limited scope of this demo project, the author did not experience any major issues or did not feel restricted by the language itself. With a more wide-spread adoption and real-world usage, those issues will hopefully be solved and overcome. Under this impression and seeing how simple it was for the author to adopt the language, he could really see the wish of Jordan Walke become reality. Reason may be able to take React or even web development to the next level.

REFERENCES

1. GitHub. 2018. Annotating JavaScript for the Closure Compiler. Date of retrieval 19.05.2018. https://github.com/google/closure-compiler/wiki/Annotating-JavaScript-for-the-Closure-Compiler

2. ECMA International. 2011. ECMA-262 Edition 5.1, The ECMAScript Language Specification. Date of Retrieval 06.05.2018. https://www.ecma-international.org/ecma-262/5.1/#sec-15.4.4.19

3. ECMA International. 2011. ECMA-262 6th Edition, The ECMAScript 2015 Language Specification. Date of Retrieval 06.05.2018. https://www.ecma-international.org/ecma-262/6.0/#sec-arrow-function-definitions

4. OCaml.org, 2018. OCaml. Date of retrieval 04.05.2018. http://ocaml.org/

5. GitHub. 2018. Flow. Date of retrieval 18.05.2018. https://github.com/facebook/flow

6. Reactiflux. 2017. Cheng Lou. Date of retrieval 05.05.2018. https://www.reactiflux.com/transcripts/cheng-lou/

7. Reason. 2018. What and Why. Date of retrieval 06.05.2018. https://reasonml.github.io/docs/en/what-and-why.html

8. Reactiflux. 2017. Jordan Walke. Date of retrieval 05.05.2018. https://www.reactiflux.com/transcripts/jordan-walke/

9. Hoare, Tony. 2009. Null References: The Billion Dollar Mistake. Date of retrieval 05.05.2018. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

10. Rauschmayer, A. 2018. Polymorphic variants.
Date of retrieval 06.05.2018.
http://reasonmlhub.com/exploring-reasonml/ch_polymorphic-vari-ants.html

11. GitHub. 2018. ECMAScript Pattern Matching.
Date of retrieval 05.05.2018.
https://github.com/tc39/proposal-pattern-matching

12. Wikipedia. 2018. Foreign function interface. Date of retrieval 06.05.2018.
https://en.wikipedia.org/wiki/Foreign_function_interface

13. Reason. 2018. Comparison to OCaml. Date of retrieval 04.05.2018.
https://reasonml.github.io/docs/en/comparison-to-ocaml.html

14. Reason. 2018. Playground. Date of retrieval 18.05.2018.
https://reasonml.github.io/en/try.html

15. Wikipedia. 2018. Conway's Game of life. Date of retrieval 08.05.2018.
https://en.wikipedia.org/wiki/Conway's_Game_of_Life

16. FreeCodeCamp. 2018. Build the Game of life.
Date of retrieval 05.05.2018.
https://www.freecodecamp.org/challenges/build-the-game-of-life

17. Redux. 2018. Read Me. Date of retrieval 18.05.2018.
https://redux.js.org/

18. GitHub. 2018. Official ReasonReact Migration Script.
Date of retrieval 18.05.2018.
https://github.com/chenglou/upgrade-reason-react

19. GitHub. 2018. Reason-apollo. Date of retrieval 19.05.2018.
https://github.com/apollographql/reason-apollo

20. OCaml Labs. 2017. PPX. Date of retrieval 20.05.2018.
http://ocamllabs.io/doc/ppx.html

21. White, L. 2015. Modular Implicits. Date of retrieval 20.05.2018. https://arxiv.org/pdf/1512.01895.pdf

22. Rauschmayer, A. 2018. What is planned for ReasonML?. Date of retrieval 05.05.2018. http://reasonmlhub.com/exploring-reasonml/ch_future.html

23. GitHub. 2018. Multicore OCaml. Date of retrieval 18.05.2018. https://github.com/ocamllabs/ocaml-multicore

APPENDICES

The following appendices can also be found online at https://github.com/matthi-askern/reason-game-of-life.

src/SharedTypes.re

```
type status =
  | Alive
  | Dead;

type cell = {status};

type row = array(cell);

type cells = array(row);

type size = (int, int);

type position = (int, int);
```

src/Logic.re

```
open SharedTypes;

let getInitialSize = () : size => {
  let width = Utils.viewportWidth;
  width > 992 ? (50, 70) : width > 576 ? (30, 50) : (15, 25);
};

/* Initialize random module */
Random.self_init();

let alivePercentile = 8;

let biggerThanAlivePercentile = num => num > alivePercentile;

let randomStatus = () : status => {
  let isAlive = biggerThanAlivePercentile(Random.int(11));
  isAlive ? Alive : Dead;
};

let randomCell = _el : cell => {status: randomStatus()};

let deadCell = _el : cell => {status: Dead};

let generateCells = (size: size, fn: _ => cell) : cells => {
  let (rows, cols) = size;
  Array.(make(cols, None) |> make(rows) |> map(map(fn)));
};

let generateEmptyCells = (size: size) => generateCells(size,
deadCell);

let generateRandomCells = (size: size) => generateCells(size,
randomCell);

let mapCells = (fn: (position, cell, cells) => cell, cells) : cells =>
  Array.(
    mapi(
      (y, row) => row |> mapi((x, cell') => fn((x, y), cell', cells)),
      cells,
    )
  );

let cycleCell = cell : cell =>
  switch (cell.status) {
  | Alive => {status: Dead}
  | Dead => {status: Alive}
  };

let toggleCell = ((x, y): position) =>
  mapCells(((x', y'), cell, _) =>
    x === x' && y === y' ? cycleCell(cell) : cell
  );

let correctIndex = (length: int, i: int) : int =>
  i === (-1) ? length - 1 : i === length ? 0 : i;

let findCell = (cells, (x, y): position) : cell => {
  let lengthX = Array.length(cells[0]);
  let lengthY = Array.length(cells);
  let x' = correctIndex(lengthX, x);
  let y' = correctIndex(lengthY, y);
  cells[y'][x'];
};

let getNeighborCells = ((x, y): position, cells) : list(cell) =>
  [
    (x - 1, y - 1),
    (x - 1, y),
    (x - 1, y + 1),
    (x, y - 1),
    (x, y + 1),
    (x + 1, y - 1),
    (x + 1, y),
    (x + 1, y + 1),
  ]
  |> List.map(findCell(cells));

let getAliveNeighbors = (cells, position) : int => {
  let neighborCells = getNeighborCells(position, cells);
  neighborCells |> List.filter(({status}) => status == Alive) |>
List.length;
};

let checkCell = (position, cell, cells) : cell => {
  let neighbors = getAliveNeighbors(cells, position);
  switch (cell.status) {
  | Alive when neighbors > 3 || neighbors < 2 => {status: Dead}
  | Dead when neighbors == 3 => {status: Alive}
  | _ => cell
  };
};

let evolution = cells : cells => mapCells(checkCell, cells);
```

src/Utils.re

```
let strE = ReasonReact.string;

[@bs.val]
external requestAnimationFrame : (unit => unit) => int =
  "requestAnimationFrame";

[@bs.val] external cancelAnimationFrame : int => unit =
"cancelAnimationFrame";

[@bs.val]
external viewportWidth : int = "document.documentElement.clientWidth";
```

src/App.re

```
ReactDOMRe.renderToElementWithId(<App />, "root");
```

src/Main.re

```
open SharedTypes;

type state = {
  size,
  generation: int,
  cells,
  timer: ref(option(int)),
  isPlaying: bool,
};

type action =
  | Evolution
  | Start
  | Stop
  | Clear
  | ToggleCell(position)
  | Random;

let initialSize = Logic.getInitialSize();

let initialState = () => {
  size: initialSize,
  generation: 0,
  cells: Logic.generateRandomCells(initialSize),
  timer: ref(None),
  isPlaying: false,
};

let component = ReasonReact.reducerComponent("App");

type self = ReasonReact.self(state, ReasonReact.noRetainedProps,
action);

let clearTimerAndStop = (self: self) => {
  switch (self.state.timer^) {
  | None => ()
  | Some(timeout) => Utils.cancelAnimationFrame(timeout)
  };
  self.send(Stop);
};

let togglePlay = (self: self, _) =>
  if (self.state.isPlaying) {
    clearTimerAndStop(self);
  } else {
    let rec play = () => {
      self.state.timer := Some(Utils.requestAnimationFrame(play));
      self.send(Evolution);
    };
    play();
    self.send(Start);
  };

let make = _children => {
  ...component,
  initialState: () => initialState(),
  reducer: (action, state) =>
    switch (action) {
    | Start => ReasonReact.Update({...state, isPlaying: true})
    | Evolution =>
      ReasonReact.Update({
        ...state,
        cells: Logic.evolution(state.cells),
        generation: state.generation + 1,
      })
    | Stop =>
      ReasonReact.Update({...state, isPlaying: false, timer:
ref(None)})
    | ToggleCell(position) =>
      ReasonReact.Update({
        ...state,
        cells: Logic.toggleCell(position, state.cells),
      })
    | Clear =>
      ReasonReact.UpdateWithSideEffects(
        {
          ...state,
          cells: Logic.generateEmptyCells(initialSize),
          generation: 0,
        },
        clearTimerAndStop,
      )
    | Random =>
      ReasonReact.UpdateWithSideEffects(
        {
          ...state,
          cells: Logic.generateRandomCells(state.size),
          generation: 0,
        },
        clearTimerAndStop,
      )
    },
  render: self =>
    <main>
      <Header />
      <Controls
        onRandom=(() => self.send(Random))
        onTogglePlay=(togglePlay(self))
        isPlaying=self.state.isPlaying
        onClear=(() => self.send(Clear))
        generation=self.state.generation
      />
      <Board
        cells=self.state.cells
        onToggle=((y, x) => self.send(ToggleCell((x, y))))
      />
      <footer>
        <a
          href="https://github.com/matthiaskern/reason-game-of-life"
          style=(ReactDOMRe.Style.make(~float="right",
~fontSize="17px", ()))
          target="_blank">
          (Utils.strE("GitHub"))
        </a>
      </footer>
    </main>,
};
```

src/Header.re

```
open Utils;

let component = ReasonReact.statelessComponent("Header");

let make = _children => {
  ...component,
  render: _self =>
    <header>
      <h1> (strE("Reason game of life")) </h1>
      <h2>
        (strE("An implementation of "))
        <a
          href="https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life"
          target="_blank">
          (strE("Conway's Game of Life"))
        </a>
        (strE(" in "))
        <a href="https://reasonml.github.io" target="_blank">
          (strE("ReasonML"))
        </a>
        (strE(" and "))
        <a href="https://reasonml.github.io/reason-react/"
target="_blank">
          (strE("ReasonReact"))
        </a>
      </h2>
    </header>,
};
```

src/Board.re

```
open SharedTypes;

let component = ReasonReact.statelessComponent("Board");

let makeCell = (onToggle, x: int, cell) =>
  <Cell key=(string_of_int(x)) cell onToggle=((_) => onToggle(x)) />;

let make = (~cells: cells, ~onToggle, _children) => {
  ...component,
  render: _self =>
    <section>
      (
        Array.mapi(
          (y, row) =>
            <div className="row" key=(string_of_int(y))>
              (
                row |> Array.mapi(makeCell(onToggle(y))) |>
ReasonReact.array
              )
            </div>,
          cells,
        )
        |> ReasonReact.array
      )
    </section>,
};
```

src/Cell.re

```
open SharedTypes;

type retainedProps = {cell};

let component =
ReasonReact.statelessComponentWithRetainedProps("Cell");

let classNameOfStatus = status : string =>
  switch (status) {
  | Alive => "alive"
  | Dead => "dead"
  };

let make = (~onToggle, ~cell: cell, _children) => {
  ...component,
  retainedProps: { cell: cell },
  shouldUpdate: ({oldSelf, newSelf}) =>
    oldSelf.retainedProps.cell.status !==
newSelf.retainedProps.cell.status,
  render: _self =>
    <div
      className=("cell " ++ classNameOfStatus(cell.status))
      onClick=((_) => onToggle())
    />,
};
```

src/Header.re

```
open Utils;

let component = ReasonReact.statelessComponent("Header");

let make = _children => {
  ...component,
  render: _self =>
    <header>
      <h1> (strE("Reason game of life")) </h1>
      <h2>
        (strE("An implementation of "))
        <a
          href="https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life"
          target="_blank">
          (strE("Conway's Game of Life"))
        </a>
        (strE(" in "))
        <a href="https://reasonml.github.io" target="_blank">
          (strE("ReasonML"))
        </a>
        (strE(" and "))
        <a href="https://reasonml.github.io/reason-react/"
target="_blank">
          (strE("ReasonReact"))
        </a>
      </h2>
    </header>,
};
```

src/Controls.re

```
open Utils;

let component = ReasonReact.statelessComponent("Controls");

let make =
    (~onRandom, ~onTogglePlay, ~isPlaying, ~onClear, ~generation,
_children) => {
  ...component,
  render: _self =>
    <aside>
      <button onClick=((_) => onRandom())> (strE("Random")) </button>
      <button onClick=((_) => onTogglePlay())>
        (strE(isPlaying ? "Stop" : "Start"))
      </button>
      <button onClick=((_) => onClear())> (strE("Clear")) </button>
      <span
        style=(
          ReactDOMRe.Style.make(
            ~float="right",
            ~color="white",
            ~fontSize="17px",
            (),
          )
        )>
        (strE("Generation: " ++ string_of_int(generation)))
      </span>
    </aside>,
};
```

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Reason game of life</title>
  <link rel="stylesheet" type="text/css" href="./styles.css">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
</head>
<body>
  <div id="root"></div>
  <script src="./bundle.js"></script>
</body>
</html>
```

public/styles.css

```
body {
  padding: 0;
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto,
Helvetica, Arial, sans-serif, 'Apple Color Emoji', 'Segoe UI Emoji',
'Segoe UI Symbol';
}

#root {
  display: flex;
  align-items: center;
  justify-content: center;
  background: linear-gradient(135deg, #fad961 0%, #f76b1c 100%);
  height: 100%;
  min-height: 100vh;
}

main {
  display: flex;
  justify-content: space-between;
  flex-direction: column;
  width: 80%;
  height: 80%;
}

main > *{
  margin-top: 0.5rem;
}

aside {
  background-color: rgba(0, 0, 0, 0.1);
  border-radius: 0.25rem;
  padding: 1rem;
}

.row {
  display: flex;
}
```

```css
.cell {
  font-size: 9px;
  width: 11px;
  height: 11px;
  margin: 0;
}

.cell:not(:last-child) {
  border-right: 1px solid black;
}
.row:not(:first-child) .cell {
  border-top: 1px solid black;
}

.dead {
  background: rgba(65, 65, 65, 0.5);
}

.alive {
  background: white;
}

section {
  display: flex;
  margin-left: 2rem;
  margin-right: 2rem;
  flex-direction: column;
  align-items: center;
}

a {
  color: white;
  text-decoration: none;
  border: 1px solid transparent;
}

a:hover {
  border: 1px dashed white;
}

button {
  background-color: transparent;
  border-width: 1px solid;
  border-color: white;
  border-style: solid;
  padding: 0.25rem 0.5rem;
  color: white;
  border-radius: 0.25rem;
  margin-right: 0.5rem;
  cursor: pointer;
}

button:focus,
button:hover,
button:active{
  border-style: dashed;
}
```

bsconfig.json

```json
{
  "name" : "reason-game-of-life",
  "reason" : {"react-jsx" : 2},
  "bs-dependencies": ["reason-react"],
  "sources": {
    "dir" : "src",
    "subdirs" : true
  },
  "refmt": 3
}
```

webpack.config.js

```js
module.exports = {
  mode: 'development',
  entry: './lib/js/src/Main.js',
  output: {
    path: __dirname + '/public',
    filename: 'bundle.js',
  },
};
```

package.json

```json
{
  "name": "reason-game-of-life",
  "scripts": {
    "start": "bsb -make-world -w",
    "build": "bsb -make-world",
    "bundle": "webpack -w",
    "clean": "bsb -clean-world"
  },
  "version": "1.0.0",
  "license": "MIT",
  "dependencies": {
    "react": "^16.2.0",
    "react-dom": "^16.2.0",
    "reason-react": "^0.4.0"
  },
  "devDependencies": {
    "bs-platform": "^3.0.0",
    "webpack": "^4.1.1",
    "webpack-cli": "^2.0.10"
  }
}
```

License