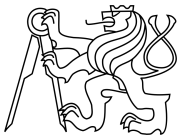
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Functional Programming for Web Frontend
Student:	Jan Luxemburk
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2017/18

Instructions

- Perform a brief review of programming paradigms.
- Perform a review of the functional programming (FP) paradigm.
- Perform a brief review of the selected FP languages: Haskell, ClojureScript, JavaScript.
- Perform a review of the state of the art of Functional Reactive Programming (FRP).
- Formulate an analysis of differences between Elm's and Javascript's libraries and development tools.
- Develop and test a sample Elm application.
- Comment your results.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague November 8, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Functional Programming for Web Frontend

Jan Luxemburk

Supervisor: Ing. Robert Pergl, Ph.D.

10th May 2017

Acknowledgements

I would like to thank my supervisor Ing. Rober Pergl, Ph.D. for his guidance and helpful approach.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 10th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Luxemburk, Jan. *Functional Programming for Web Frontend.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstract

Primary focus of this thesis is the programming language Elm. Elm is thoroughly investigated; on a theoretical level, in the context of functional programming concepts like function purity, side effects, and immutability. From a practical point of view, I review the quality of Elm's tooling and its libraries, and I compare them to JavaScript. A sample Elm application is also part of the thesis.

Keywords Elm, functional programming, frontend development, JavaScript

Abstrakt

Tato práce se věnuje novému programovacímu jazyku Elm. Elm je důkladně prozkoumán. Na teoretické úrovni, v kontextu konceptů functionálního programovaní jako jsou "pure" funkce, "side" efekty nebo perzistentní datové struktury. Z praktického pohledu se věnuji dostupným vývojářským nástrojům a knihovnám. Součástí práce je i ukázková Elm aplikace.

Klíčová slova Elm, funkcionální programování, frontend developement, JavaScript

Contents

List of Figures

Introduction

Functional programming has its roots in Lambda calculus, a mathematical formal system from the 1930s. It was mostly used by academia and was considered hard to learn. This is changing with the rise of the programming language JavaScript. JavaScript is the core of modern web pages and it is the only programming language that is supported without plugins by all modern web browsers. Consequently, it is used everywhere on the web. Because JavaScript encourages a functional style of programming, more and more people get to know it; and they soon find the advantages that this style brings. These are particularly better code modularity, reusability, and expressiveness.

Nevertheless, JavaScript is not flawless; particularly, it gives a programmer too much freedom. One solution is to use other, better-designed programming languages that are then compiled to JavaScript. Programming language Elm is one of these languages.

Chapter 1

Goals and Approach

1.1 Goals

This thesis has three goals: (1) to perform a review of functional programming, its languages, and especially the Elm language, (2) to develop a sample Elm application, with a focus on concepts of functional programming, and (3) to formulate an analysis of differences between Elm's and JavaScript's development tools and libraries.

Research questions of this thesis are:

1. How does Elm implement the key concepts of functional programming?
2. Are Elm and its development tools ready for real-world use?

1.2 Methodology and Structure

I start with reviewing programming paradigms, chiefly focusing on the functional one; then I introduce functional programming languages that influenced the Elm language and are important for web development. The rest of Review Chapter is about the Elm language. In Chapter Application in Elm, I perform a brief analysis of selected domain and discuss the implementation, architecture, and testing of the application. I use UML 2 to model the application. In the last chapter, I use experience gained by working on the sample application and my experience with JavaScript web development to formulate an analysis of differences between Elm's and JavaScript's development tools and libraries.

Chapter 2

Review

2.1 Programming paradigms

There is a huge amount of problems in programming and solving them requires different techniques. That is why we have hundreds of programming languages, each of them suitable for specific tasks. Languages vary in syntax and design, but some of them use similar programming concepts and style. A programming paradigm is a set of ideas, rules or concepts that a group of programming languages shares in common. For example imperative languages, languages supporting imperative paradigm, share that they *"use a sequence of statements to changes program's state"*. [1] A programming language can belong to more than one paradigm.

2.2 Declarative programming

Declarative programming languages provide a higher and more abstract level of programming, which leads to reliable and maintainable programs. [2] From a programmer point of view, the basic property is that programming is lifted to a higher level of abstraction; at this higher level of abstraction the programmer can concentrate on *what* is to be computed, not necessarily *how* it is to be computed. [3] In *Elmulating JavaScript* authors point out that a declarative approach to writing code helps manage growing complexity of web development. [4] The author of the Elm language agrees: *"User interfaces become a lot simpler to write when a lot of irrelevant details can be left to compiler."* [5]

Frans Coenen summarizes characteristics of declarative programming: [6]

- Model of computation based on a system where relationships are specified directly in terms of the input data.
- Made up of sets of definitions or equations describing relations which specify what is to be computed, not how it is to be computed.

- Order of execution does not matter (no side effects).
- Programmer no longer responsible for control.

Declarative programming includes other paradigms, for example functional programming, logic programming, or domain-specific languages (SQL, HTML).

2.3 Functional programming

Functional programming is called so because its fundamental operation is an application of a function to arguments. Functions in functional programming are called *first-class*, which means that they are treated like any other values and can be passed as arguments to other functions or be returned as a result of a function. *"A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives."* [7]

Functional programming originates in a mathematical formal system named Lambda calculus.

2.3.1 Lambda calculus

In the 1930s two distinct formal systems describing computability were introduced: Turing's Turing machines and Alonzo Church's Lambda calculus. *"Both are well defined in terms of a simple set of primitive operations and simple set of rules for structuring operations; most important, both has a proof theory."* [8] They have been shown formally to be equivalent to each other and also generalize von Neumann machines – digital computers. [8]

Basics of Lambda calculus

Lambda calculus is a surprisingly simple, yet powerful system for manipulating lambda expressions. It is based on function abstraction – to generalize expressions by introducing names, and function application – to evaluate generalized expressions by giving names proper values. [8]

Consider this lambda expression:

$$\lambda x.x$$

It is an identity function. To use this function we would use a function application to apply it on arguments – for example on itself:

$$(\lambda x.x\ \lambda x.x)$$

Generally, a λ expression can be defined as follows:

$$<expression> ::= <name> \mid <function> \mid <application>$$

A λ expression may be a name to identify a function, a function introducing an abstraction, or a function application to specialize an abstraction. [8] A λ function has the form:

$$<function> ::= \lambda<name>.<body>$$

where:

$$<body> ::= <expression>$$

The *name* is called a bound variable and is like a function parameter. *"The '.' [dot] separates the name from the expression in which abstraction with a name takes place."* [8] A λ function can have only one parameter. To achieve functions with more parameters, the body expression can be another λ function. For example:

$$\lambda first.\lambda second.first$$

This chaining of one-parameter functions is called *currying* in functional programming languages. A function application has the form:

$$<application> ::= (<function\ expression> <arguments\ expression>)$$

where:

$$<function\ expression> ::= <expression>$$
$$<arguments\ expression> ::= <expression>$$

There are two approaches to evaluating function applications. *"Either arguments are reduced to simplest terms before being passed into functions (called eager, strict, or applicative evaluation), or are passed into functions and be reduced only once that function needs the value (called normal evaluation)."* [9] In both cases, the function expression is evaluated to return a function. All occurrences of the function's bound variable in the function's body expression are then replaced by a value of the argument expression or by the unevaluated argument expression – depending on the evaluation order. Finally, the function's body expression is evaluated. [8]

Formally, the replacement of the bound variable with an argument in a function's body is called a *beta reduction*. It is one of three reduction rules that Lambda calculus defines for simplifying lambda expressions without changing their value. [9]

Evaluation strategy

Evaluation strategy is an important characteristic of programming languages. Most of programming languages use strict evaluation. There are more versions of strict evaluation, e.g., call-by-value or call-by-reference. Haskell, on the other hand, utilizes a type of normal evaluation called *lazy evaluation*. [10]

Functional programming languages use a number of concepts that come from Lambda calculus. Some of them are reviewed in the following sections.

2.3.2 Higher-order functions

A higher-order function is a function that takes other functions as arguments or returns a function. [11] *"Functional languages allow functions that are indivisible in conventional programming languages to be expressed as a combination of parts — a general higher-order function and some particular specializing functions. Once defined, such higher-order functions allow many operations to be programmed very easily."* [7] John Hughes also advocates that modularity, which is highly increased by higher-order functions, is the key for successful and efficient programming. [7]

Common use of the higher-order functions is in data structures processing. Three most used higher-order functions for a list and array manipulation are *map*, *filter*, and *reduce*. They are even implemented in non-functional languages such as Java [12], C++ [13], PHP [14], or Python [15].

Map

A mapping function applies a function on each element of a list and returns the list of these applications.

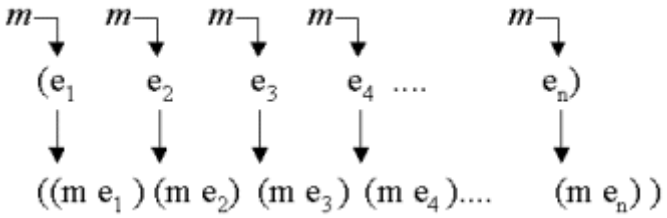


Figure 2.1: Mapping a function m on a list $(e_1, e_2, \ldots e_n)$ [16]

Filter

A filtering function applies a predicate (a boolean function) on every element of a list. Only elements for which the predicate returns true are returned in a new list. [16]

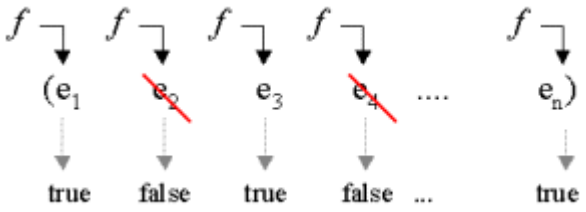


Figure 2.2: Filtering a list with a predicate f [16]

Reduce

Reduce, also known as fold, is a function for getting a single value by operating on a list of values. *"Fold is a family of higher order functions that process a data structure in some order and build a return value."* [17] The result value can also be another list. Map and filter functions can be implemented with the reduce function. [18]

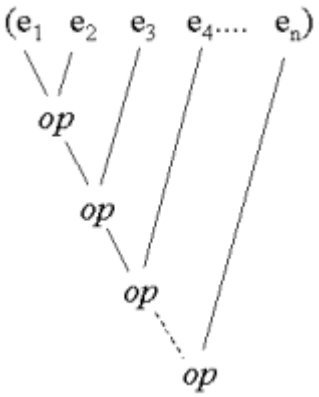


Figure 2.3: Reduction of a list [16]

2.3.3 Currying

Currying is a technique of transforming a function that takes multiple arguments into series of functions that take parts of the arguments by one. In Haskell, *"All functions are considered curried: That is, all functions take just one argument."* [19] Curry function signature:

$$curry : ((a,b) \rightarrow c) \rightarrow (a \rightarrow (b \rightarrow c))$$

where:

$$(a,b) \rightarrow c$$

is the uncurried version of a function; it takes a and b as arguments and returns c, and:

$$a \rightarrow (b \rightarrow c)$$

is the curried version that takes only one argument a, and returns a function with signature $b \rightarrow c$. Both forms are equally expressive [19], and when we supply two arguments to either curried or uncurried version of the function, the result is always the same. The difference is that curried functions allow a use of partial application, which is a process of fixing a number of arguments to a function or in other words: *"Passing less than the full number of arguments to a function that takes multiple arguments."* [20]

2.3.4 Control flow and side effects

A function or an expression is said to have side effects if it modifies some state outside of its scope or if it has any observable interaction with its calling function or outside world. [21] Functions in a mathematical sense only map an input to an output. Any other behavior than mapping an input to an output, which is in the context of programming any other behavior than computing return value from supplied arguments, is called a side effect. Examples of side effects are modifying a global variable or an argument, raising an exception, writing data or reading a file.

John Hughes in [7] summarizes advantages of eliminating side effects: *"Since no side-effect can change an expression's value, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa — that is, programs are referentially transparent. This freedom helps make functional programs more tractable mathematically than their conventional counterparts."* This mostly applies to pure functional languages – those which completely prohibit side effects, for example Haskell. [11] Other non-pure functional languages do not restrict side effects, but it is customary for programmers to avoid them. Because impure code is more prone to errors, it is usually beneficial to write a significant part

of a program in a purely functional fashion, without side effects, and keep the code involving state and input/output to the minimum. [11]

Redux, a JavaScript frontend library, separates a state mutation from the rest of codebase; as result, it lowers code complexity and makes the state mutation predictable. Redux also use immutable data structures and pure functions. [22] This is one of many examples how functional programming influences current web development technologies.

2.3.5 Pure functions

A function is called pure if: (1) it does not cause side effects, (2) its return value is not affected by side effects, and (3) given the same arguments, it returns the same value each time it is evaluated. Examples of pure functions are mathematical functions or a function *length(s)* returning a length of the string *s*. Any function accessing a global variable is not pure because the value of the global variable can change during program execution and thus the function's return value can differ between evaluations.

2.3.6 Immutable data structures

Purely functional programs typically operate on immutable data. [11] Immutability is achieved by using persistent data structures. A persistent data structure is a *"data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure"*. [23] Common parts of the original and modified structures are often shared, which saves memory. [11]

2.4 Functional programming languages

Ever since Lambda calculus was introduced in the 1930s, new functional languages continue to emerge. Two important languages of the 20th century are Lisp and ML (MetaLanguage). Lisp was created in 1958 and is the second oldest programming language. [24] A prominent feature of Lisp and its dialects is that data and program (source code) share common structure – a linked list. [8] ML originated in the mid-1970s as a language for building proofs in a formal reasoning system. [8] ML's most known dialects are Standard ML and Caml. In ML was for the first time used the Hindley–Milner type system, whose type inference algorithm can automatically assign types of most expressions without requiring explicit type annotations. [25]

I briefly review functional languages that are relevant to frontend development. Elm is reviewed in its own Section 2.6.

2.4.1 Haskell

Haskell is a general-purpose, statically typed, lazy-evaluated, purely functional language, *"incorporating many recent innovations in programming language design"*. [26] [11] Haskell is standardized and the Glasgow Haskell Compiler is its main implementation. It supports higher-order functions, powerful abstractions, user-defined algebraic types, pattern matching and much more. [26]

Abstractions

The key feature of programming languages is their capability for designing abstractions (an example of an abstraction that almost all languages support is a function). Haskell tends to use more rigorously defined mathematical abstractions, some examples are Monad, Applicative, Functor, Monoid. Haskell represents these abstracts through type classes along with a set of laws that their instances must obey. [27] Once a programmer learns these abstractions, they help him in solving various programming challenges.

Monads

Monads come from a mathematical theory called Category theory. In Haskell, monads are used for example for handling input/output, state, exceptions, and nullability. [28] Monads can be thought as value containers with two important functions, *return* and *bind*. The *return* function creates a monad by wrapping a value and the *bind'* function serves to compose monads with functions or other monads.

IO monad *"Input/Output is incompatible with a pure functional language because it is not referentially transparent and side-effect free."* [29] A function cannot directly cause a side effect but it can construct a monad describing that side effect. [30] IO monad holds an IO action, a description of an input/output operation. The IO actions are *"combined within the IO monad, in a purely functional manner, to create more complex IO actions"*. [29] The Haskell's runtime system performs these operations at a convenient time and if necessary, returns values, for example when reading a file. This is how Haskell separates pure and impure, by representing side effects as pure values, combining them in a functional manner and actually performing them "outside" of Haskell. [31]

Pattern matching

Pattern matching enables a programmer to specify different function bodies for different arguments using patterns. When a function is called, the appropriate body is chosen by comparing the actual arguments with the various patterns; the form and/or the content of the arguments is compared. [32] After a

case (function's body) is selected, the argument or parts of it may be assigned to variables in the selected body. Example use of pattern matching is the following function for computing the length of a list (":" is an infix operator for list concatenation).

listlength :: [a] -> Integer
listlength [] = 0
listlength (first:rest) = 1 + *listlength* rest

where the first line is a type signature telling that the function *listlength* accepts a list and return an integer. The list can contain values of any type. The function *listlength* has two cases, for an empty and non-empty list. The pattern "[]" matches an empty list, and zero is returned. The pattern "(first:rest)" matches any non-empty list – then the list without first element is assigned to variable *rest* and the function is called recursively. Patterns in this example have the form of a list; another widely used forms of patterns are tuples or algebraic data types.

Algebraic data types

In Haskell and other languages featuring algebraic data types, it is easy and straightforward for a programmer to introduce new data types. An algebraic data type is a composite type, i.e., a type formed by combining other types. [33] Two common classes of algebraic types are product types and sum types, hence the name "algebraic".

A product type is often called a tuple type since it is *"essentially just a cartesian product of other types"*. [34] On the other hand, a sum type represents an alternation. A sum type has several different but fixed types and only one of the types can be used at a time. *"It can be thought of as a type that has several cases, each of which should be handled correctly when that type is manipulated."* [33] It is important that *"sums and products can be repeatedly combined into arbitrarily large structures"*. [35] Another name for a sum type is a union type.

Pattern matching is used for manipulating algebraic data types, e.g., if a function accepts a sum type, it uses patterns for matching different cases of that type. If a programmer forgot to handle any case, the compiler would warn him.

2.4.2 JavaScript

JavaScript is an interpreted, dynamically typed, high-level programming language. [36] It has been standardized in the ECMAScript language specification and its main interpreters, also called JavaScript engines, are V8 (Chrome, Node.js) and SpiderMonkey (Mozilla Firefox). [37] The specification is developed by a committee of browser vendors and other big tech companies.

The latest version of JavaScript is ECMAScript 7; ECMAScript 8 is currently being developed. [38] Because not all web browsers support all new features of the latest version, it is common practice to transpile source code to the older version of JavaScript (this is possible with most new features but not all of them). JavaScript supports an object-oriented, imperative and functional style of programming.

Object-oriented JavaScript has prototype-based inheritance and objects are extensively used throughout the language. Even functions are by type objects. Classes were added in ECMAScipt 5 but only as "syntactic sugar", internally the inheritance remains prototypal. [39]

Functional JavaScript functions are *first-class* – they are treated as values and can be passed as arguments or returned by other functions. When functions are moved around, they keep the connection to the variables of their surrounding scopes and they are called *closures*. [39] Since ECMAScipt 5, JavaScript supports *arrow* functions, more concise syntax for function expressions. There is no built-in support for currying but it is possible to implement own curry function. Map, reduce and filter functions are defined for arrays and a programmer is free to introduce other higher-order functions. Immutable data structures can be supported with third-party libraries. [40] There are no restrictions regarding side effects. Overall, JavaScript encourages a functional style of programming.

Imperative JavaScript includes many of the imperative programming constructs, e.g., *if-else* statement, *while* and *for* loops, *switch* statement, or *try/catch/finally* statement for handling the exceptions.

2.4.3 Clojure & ClojureScript

Clojure is a dialect of Lisp programming language. It is a functional language promoting immutability and designed for concurrency. [41] ClojureScript is a compiler for Clojure that targets JavaScript. *"ClojureScript is designed to be a "guest" language. This means that the language works well on top of an existing ecosystem such as JavaScript."* [42] ClojureScript inherits, with some exceptions [43], properties of Clojure, plus it has a JavaScript interop that enables consuming arbitrary JavaScript code and mapping values from JavaScript to ClojureScript, and vice versa.

2.4.4 PureScript

PureScript is a Haskell-like language that compiles to JavaScript. Unlike Haskell, PureScript is strictly evaluated (because JavaScript is strict). Its most

notable feature is that it offers high-level abstractions (the same as Haskell, for example type classes). [44]

2.5 Functional reactive programming

Functional reactive programming (FRP) is a subset of both functional and reactive programming. A reactive program is event-based, acts in response to input, and is viewed as a flow of data instead of the traditional flow of control. [45] In FRP, dynamic time-varying values are *first-class*; they can be combined together and passed into and out of functions. Time-varying values are called *behaviors* and Conal Elliott, one of FRP authors, describe them: *"Behaviors are built up out of a few primitives, constant (static) behaviors and time (clock), and then with sequential and parallel combination. n behaviors are combined by applying an n-ary function (on static values), point-wise, i.e., continuously over time."* [46] To account for discrete phenomena, FRP has *events*. An *event* has a stream of occurrences and each occurrence has a time and a value. [46] In recent formulations of FRP, ideas of *behaviors* and *events* are combined into *signals*.

Reactive program

An example of how a simple reactive program looks like, in pseudo-code:

$$x = <mouse\text{-}x>$$
$$y = <mouse\text{-}y>$$

In imperative programming, these lines would be executed only once. The coordinates of the mouse would be assigned to x and y, but the moment the mouse moves, x and y would no longer hold the correct position. In FRP, however, x and y are "synced" and always have the current mouse position. The underlying implementation of FRP is ensuring that. To extend the program:

$$minX = x - 10$$
$$minY = y - 10$$
$$maxX = x + 10$$
$$maxY = y + 10$$

We have now defined new *behaviors*, time-varying values, and we can combine them into:

$$rectangle(minX, minY, maxX, maxY)$$

This way, a rectangle box is drawn around the mouse pointer all the time whenever the mouse moves.

Implementations

Functional reactive programming is implemented as a lightweight library in many programming languages. [45]

2.6 Elm language

Evan Czaplicki designed Elm in his thesis *"Elm: Concurrent FRP for Functional GUIs"* in 2012. Elm is a functional programming language for graphical user interfaces targeting the most widespread GUI platform – the web. [5] It compiles to JavaScript, HTML, and CSS, and can be therefore run in any modern web browser. Elm has a *"very strong emphasis on simplicity, ease-of-use, and quality tooling"* [47] which, combined with a far-reaching web platform, makes Elm good choice for functional programming beginners. Elm has several features reviewed earlier in this thesis: (1) all data structures are immutable, (2) functions take one argument and are curried, and (3) side effects are managed by the Elm runtime. In the following sections, I review Elm's syntax, operators, data structures and common practices. JavaScript is sometimes used for comparison.

2.6.1 Syntax

Elm's syntax is similar to Haskell's. When compared to JavaScript's, the main differences are: (1) indentation is used for blocks and function bodies instead of curly brackets *"{}"*, (2) in function calls, parentheses are omitted and arguments are separated by spaces, (3) no semicolon is necessary for terminating statements, and (4) function bodies contain implicit return. This more concise style is common in functional languages but could be confusing for beginners.

2.6.2 Type system

Type system is one of the Elm's most recognized features, not because it brings anything extraordinary new (in fact, it is quite simple in comparison to Haskell), but rather that JavaScript is dynamically typed and many web developers have no experience with static type system. Elm uses static type checking and type inference. Static type checking is *"the process of verifying the type safety of a program based on analysis of a program's text (source code)"* [48], and with type-inference, most expressions don't have to be explicitly type-annotated; rather Elm's compiler analyzes source code and deduces all necessary types.

Type annotations

Although type annotations are voluntary, it is customary to annotate all functions. By writing type annotations to functions, a programmer expresses his intent and it is a kind of code documentation. The compiler compares the annotated type with the inferred type and shows a warning if they differ.

```
1 connectWords : String -> String -> String
2 connectWords firstWord secondWord =
3   firstWord ++ secondWord
```

Listing 2.1: Example of a type annotation in Elm

On the first line is a type annotation that consists of a function name, a colon, and a type. The colon separates the name from the type and it can be read as "has type". Symbol of an arrow indicates a function; on the left of the arrow is the type of an argument (functions are curried and accept one argument); on the right is the return type. *connectWords* is a function that accepts String and returns other function with type annotation of "String -> String". An imprecise interpretation would be that it accepts two Strings and returns a String. A programmer can annotate other values than functions, for example tuples, let expressions, or records.

Union types & type constructors

Union types, also called tagged unions, are similar to *sum algebraic data types* explained in the Haskell Section 2.4.1. Their purpose is to model complex data naturally. [47] Union types are tightly coupled with a *case-of* expression, which is a pattern matching mechanism for consuming union types and data structures.

```
1 type Day = Mon | Tue | Wen | Thu | Fri | Sat | Sun
2
3 dayString = Day -> String
4 daString aDay =
5   case aDay of
6     Mon -> "Monday"
7     Thu -> "Tuesday"
8     ...
```

Listing 2.2: Example of a union type in Elm

This snippet shows how a union type works together with the *case-of* expression. The keyword *type* is used for introducing new types. In the first line, type options for the type *Day* are listed on the right side of the equals sign – *Mon*, *Tue*, *Wen*, ..., and are called *data constructors*. They are called constructors because they create values. In the *dayString* function, *case-of* is used to match *aDay* argument against all type options of the type *Day*, when a match succeeds, the appropriate string is returned.

Maybe is a built-in type that allows expressing the idea of a missing value. [49]

```
1 type Maybe a = Just a | Nothing
```

Listing 2.3: Maybe type in Elm

Two important things have changed from the previous example, (1) the *Maybe* type is declared with a *type variable* **a**, and (2) the second type option *Just* carries a payload, in other words, it wraps a value. *Just* is a data constructor accepting one argument of yet unspecified type **a**. Type variables enable to declare polymorphic types. *Maybe* itself is called a *type constructor*; it cannot be used as a type until an argument is supplied for the type variable **a** – only then is created a valid type, like *Maybe Int* or *Maybe String*.

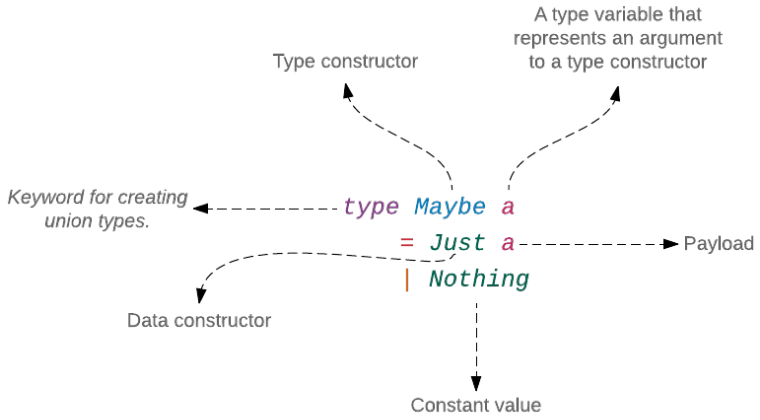


Figure 2.4: Explanation of Elm union types and type variables [50]

Type alias

The keyword *type alias* creates a new name for an existing type. It does not create a new type. Aliasing simplifies long type definitions (as shown in Section 2.6.4).

2.6.3 Operators

Elm has standard operators for arithmetics (including exponentiation and integer division), for boolean and bitwise operations and for comparing. The operator "++" appends things together, e.g., strings or lists. The operator "::" adds an element to the front of a list. Operators in Elm are like any other functions; they take two arguments and can be also used in a prefix form. It is possible to define own operators.

Function operators

Elm has operators that are special for functional programming languages. The function composition operator, "<<", is used for pipelining the result of one function to the input of another and it creates entirely new function.

```
1 (<<) : (b -> c) -> (a -> b) -> a -> c
2 (|>) : a -> (a -> b) -> b
```

Listing 2.4: Type signature of Elm functional operators

By examining the signature, the function composition operator takes two functions and it is necessary that the type of the output of the second function (type variable b) matches the type of the input of the first function (also b). This operator is equivalent to the mathematical function composition ($f \circ g$). The function composition operator exists also in the opposite direction – ">>".

The forward "|>" and backward "<|" function application operators are used for applying a function to an argument; they have lower precedence than a normal function application by adjacency. The primary use of the functional application operators is to avoid parenthesis and to write code in a more natural way. [51] Consider the following code example of creating a pentagon:

```
1 scale 2 (move (10,10) (filled blue (ngon 5 30)))
```

Listing 2.5: Creating a pentagon without the function application operator

Which can be written as:

```
1 ngon 5 30
2   |> filled blue
3   |> move (10,10)
4   |> scale 2
```

Listing 2.6: Creating a pentagon with the function application operator

2.6.4 Data structures

Elm provides multiple data structures. List, Record, and Tuple are implicitly imported and thus available in all Elm files. [52] Array, Set, and Dict are defined in the Elm's core package (the core package is like a standard library) and they have to be imported explicitly. All data structures are immutable and any mutation of a data structure yields a new one.

List

A list is a common data structure of functional languages. Conceptually, a list is divided into two parts: a head and a tail. The first element is called the head; the tail is another list representing the rest of the elements. Every element in

a list must have the same type. A list is created with square brackets and each element separated by a comma. The *List* module provides various functions for list operations, such as sorting, splitting, filtering, mapping or reducing (see 2.3.2). Lists are suitable for traversing elements in a linear order, from beginning to end. Elements in a list cannot be accessed by index.

Array

Array elements can be accessed by index and they all must have the same type. Unlike lists, arrays do not have literal for creation; they are created by functions from the *Array* module, such as *repeat*, *initialize* or *fromList*. The *Array* module also provides functions *get* and *set* for an indexed access and other functions for an array manipulation.

Tuple

A tuple can contain values of different types, but once created, the number of its elements cannot be changed. Tuples are created with parenthesis, and their elements separated by commas. The empty tuple, "()", has a special meaning in Elm; it is used as a placeholder for an empty value. The *Tuple* module contains accessing and mapping functions only for tuples of size two. It is advised to use records instead of tuples for sizes bigger than two. [53]

Record

Unlike in so far discussed data structures, values in records have associated names. Records contain key-value pairs, called *fields*, and the values can have different types. Records are created with curly brackets, fields separated by commas, and values from keys separated by equals signs. The first way to access fields of records is a standard dot notation. The second way is by a special accessing function – a dot plus field name (".title" in the following example). It is not possible to add fields to a record or remove them. The compiler checks that only the existent fields are accessed. The syntax for updating a field value is shown on the eighth line of the following example.

```
1 myBook : { title: String, author: String, pages: Int }
2 myBook =
3   { title = "The Trial", author = "Karel Capek", pages = 255 }
4
5 myBook.title -- The Trial
6 .title myBook -- The Trial
7
8 { myBook | author = "Franz Kafka" }
9 myBook.author -- Franz Kafka
```

Listing 2.7: Syntax of records in Elm

The type aliasing is convenient for simplifying record's type annotation, and also it creates a *record constructor*. [54] This is a special case – *type alias* creates a constructor only when used with records. Record constructor is a function that serves as another way of creating records. Arguments to the constructor must be supplied in the same order as their labels appear in the type alias definition.

```
1 type alias Book = { title: String, author: String, pages: Int }
2 Book : String -> String -> Int -> Book -- record constructor
3
4 myBook = Book "Man's Search for Meaning" "Viktor E. Frankl" 154
```

Listing 2.8: Type aliasing record's signature in Elm

2.6.5 Elm Architecture

The Elm Architecture is a pattern for building web applications. It is promoted by authors of Elm and also by the rest of the Elm community, and it is a standard way of creating Elm applications. Having a prescribed recipe for creating applications is not common in other programming languages. For example in JavaScript, different frameworks have different approaches for designing applications and there is a multitude of ways of solving particular problems. Elm promotes the idea of having one, correct way of doing things – this is possible chiefly because Elm specializes only for GUIs.

The architecture

Every Elm application contains these parts: (a diagram of the architecture can be found in Section 3.2)

- **Model** – the state of application
- **Update** – a way to update the state
- **View** – a way to view the state as HTML

Model A model represents the state of the application. Usually a type-aliased record is used as the model, but for simple applications, it can be a single value, e.g., string or integer. The type annotation of the model precisely describes the form of the application's state – names of fields, their types, and how they are nested.

```
1 type alias Model =
2   { blogTitle : String
3   , articles : List Article
4   , filter : Bool
5   }
```

```
6  type alias Article =
7    { title : String
8    , content : String
9    }
10
11 model : Model
12 model =
13   { blogTitle = "Example Blog"
14   , articles = []
15   , filter = False
16   }
```

Listing 2.9: Model definition (code is formatted according to standard rules)

Because the model is a record, no fields can be added to it or removed from it. Only the values of fields can change. The model's type annotation thus represents the form of the state throughout the application's execution. A model is once initialized (in this example on lines thirteen to eighteen) and after that, the only way to change it is via the *update* function.

Update Changes of a model happen through the *update* function. Under the hood, the Elm runtime invokes the *update* function with various messages. These messages come from three sources: (1) from user's interaction with the graphical interface, e.g., a user submits a form or an input field gets focus, (2) from listening for external inputs, such as keyboard events or a websocket communication, and (3) from runtime's responses to *commands*. Commands are used for impure actions, e.g., sending HTTP request or reading current time. Instead of doing such actions directly, a programmer tells the runtime to do them by returning a command from the *update* function; when the result of the command is ready, the runtime sends a message back to the *update* function.

Messages define all possible actions or events that can happen in an Elm application. They are represented as a union type; each *type option* is a message, optionally carrying data.

```
1  type Msg
2    = Roll
3    | NewFace Int
4
5  update : Msg -> Model -> (Model, Cmd Msg)
6  update msg model =
7    case msg of
8      Roll ->
9        (model, Random.generate NewFace (Random.int 1 6))
10     NewFace newFace ->
11       ({ model | diceFace = newFace }, Cmd.none)
```

Listing 2.10: Example of the update function, messages, and commands [47]

The *update* function takes a message, a model (the current state of an application), and returns a tuple of type "(Model, Cmd Msg)". The "Model" in the return tuple is the new state of the application. The "Cmd Msg" is a command for the Elm runtime. By returning a command from the *update* function, a programmer tells the runtime to execute it.

The code would be run as follows:

1. *Update* receives a *Roll* message, for example, because the user clicks on a "Roll Dice" button.
2. *Roll* pattern-matches the first case of the *case-of* expression. In the return tuple (line nine), the model is returned unchanged and a new command is created. The *Random.generate* is a function that as the first argument accepts a *data constructor* (see Section 2.6.2). The *data constructor* tells the runtime what message it should send back. In this case, the runtime will send a *NewsFace* message (line three) containing an integer.
3. The Elm runtime generates a random number, wraps it in a *NewsFace* message, and sends it to the *update* function ("sending a message to the *update* function" means invoking it with that message).
4. *Update* receives a *NewFace* message. The message matches the second pattern "NewFace newFace"; by this pattern, the wrapped integer is extracted to the variable *newFace*, which is afterward used to update the model (the model is a record with a *diceFace* field). The "Cmd.none" stands for no command.

View The view is represented by functions that create HTML markup. Their input is the model or a part of it, and their output is a markup that represents the current state of the application. Every HTML element (div, input, button, ...) has defined a corresponding function in the *Html* module. These functions have in common that they take a list of attributes as the first argument and a list of children elements as the second.

```
1  view : Model -> Html Msg
2  view model =
3    div
4      [ class "blog" ]
5      [ h1 [] [ text model.blogTitle ]
6      , p [] [ text "Welcome to my blog" ]
7      , section
8          [ class "articles" ]
9          (List.map viewArticle model.articles)
10     ]
```

Listing 2.11: Example of a view function in Elm

The Elm's style of writing HTML is different from writing normal ".html" files. In Elm, HTML is written in the same files as code; it is indistinguishable from code. It is more "powerful" because a programmer can use all language features, as in this example on the ninth line, *List.map* is used to generate HTML markup for every article.

Elm uses a virtual DOM for an optimization of HTML rendering. DOM, Document Object Model, is a tree structure that web browsers use for representing HTML documents. Changing DOM means re-rendering the page and it is a resources expensive operation. Instead of always manipulating DOM directly, the Elm runtime has a virtual DOM and change the real DOM only when necessary. According to performance benchmarks of Evan Czaplicki [55], Elm has the fastest HTML rendering among React, Angular, and Ember by roughly 120 to 150 percent.

Subscriptions

A subscription is a way of telling the Elm runtime that the application is interested in listening for some external input, for example keyboard events, mouse movements, browser location changes (changes of URL), or a websocket communication. Similarly as with *commands*, a programmer must specify what *type* of message should the runtime send to the *update* function every time a subscribed event happens.

2.6.6 JavaScript interop

Elm offers a way to communicate between JavaScript and Elm. On the Elm side, a programmer can use a keyword *port* to create a communication channel between the two runtimes. *Commands* are used for sending data from an Elm application to JavaScript, and by *subscribing* to a port, the Elm application receives data from JavaScript. On the JavaScript side, ports are available as properties of application's object – an Elm application is initialized from JavaScript and a programmer can save the reference to it. Ports in JavaScript have methods *send* and *subscribe*. Data coming from JavaScript are type validated and when invalid data are sent, an exception is thrown on the JavaScript side.

Chapter 3

Application in Elm

The purpose of creating this application is to show principles of functional programming and to demonstrate the Elm programming language. I developed a simplified email inbox (think of it as a clone of Gmail). The sample application is a single-page application (SPA) website that users visit in a web browser and use to send, receive and manage emails. A single-page application does not use extra queries to the server to download pages; all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load. After that, the page does not reload nor does control transfer to another page. All interactions and communication are handled by JavaScript in the user's web browser.

I focus on what is Elm specialized for, on the frontend. I therefore do not develop, nor design the backend of the application.

3.1 Analysis

The inbox provides a graphical interface for email communication.

3.1.1 Requirements

The main entities of this application are User, Email Address, Email, Inbox. Functional requirements are:

- A user can own multiple email addresses; if so, the inbox shows emails of all the user's addresses.
- A user can send emails, either replying to an existing conversation or starting a new one. If a user owns multiple addresses, he can choose from which he wants to send an email. A user can save an email as a draft.
- The inbox has various capabilities for management of emails, such as categorizing, searching, filtering and sorting. A user can attach labels

to emails or mark them as important. A user can search for emails by author, recipients, subject or by a combination of these.

- A user can add filters to the inbox. When an incoming email matches a filter's rule, the specified action is performed. Filter actions are: marking an incoming email as read or as spam, deleting it, or archiving it.

Non-functional requirements are:

- When a user is writing an email, the inbox saves the email in the local storage of the web browser; therefore no emails are lost in case of a network connection loss or a web browser failure.

- The application must be compatible with all standard web browsers. Responsibility for various screen sizes is required, but it is not necessary to support mobile devices.

3.1.2 Domain model

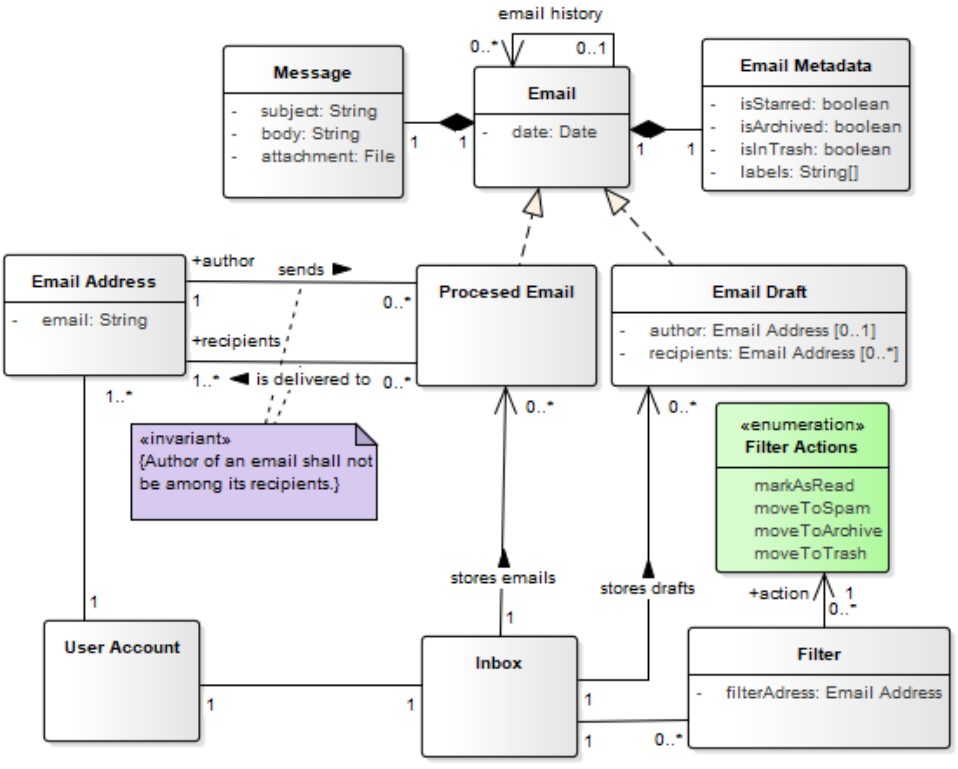


Figure 3.1: Domain model of the inbox

3.1.3 Use cases

- **UC1** Send an email (modeled in diagram below)
- **UC2** Search for emails from a particular person
- **UC3** Mark all emails in inbox as read

There is only one actor - the User, and he is associated with all use cases.

3.1.4 Activity diagram

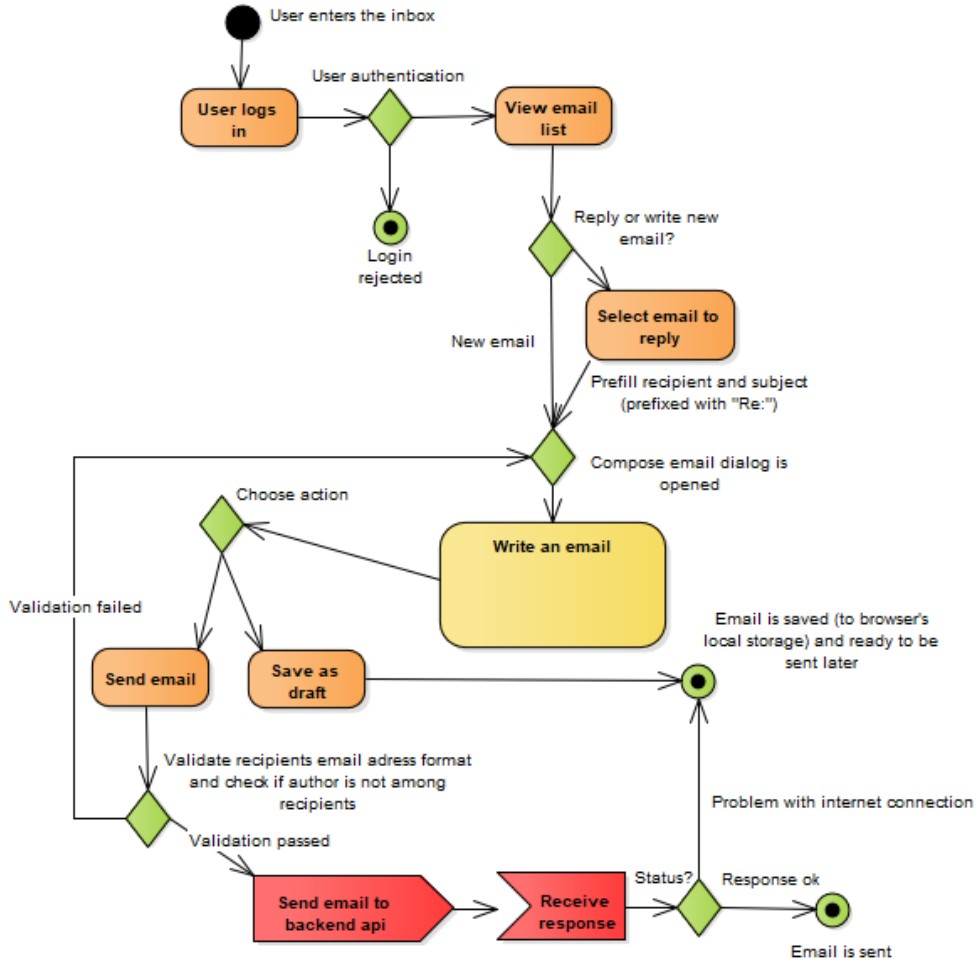


Figure 3.2: Activity diagram of sending an email

3.2 Architecture

I adhered to Mode-Update-View Elm Architecture that was discussed in Section 2.6.5. The symbol in the middle of the following diagram, which is the logo of the Elm language, stands for the Elm runtime.

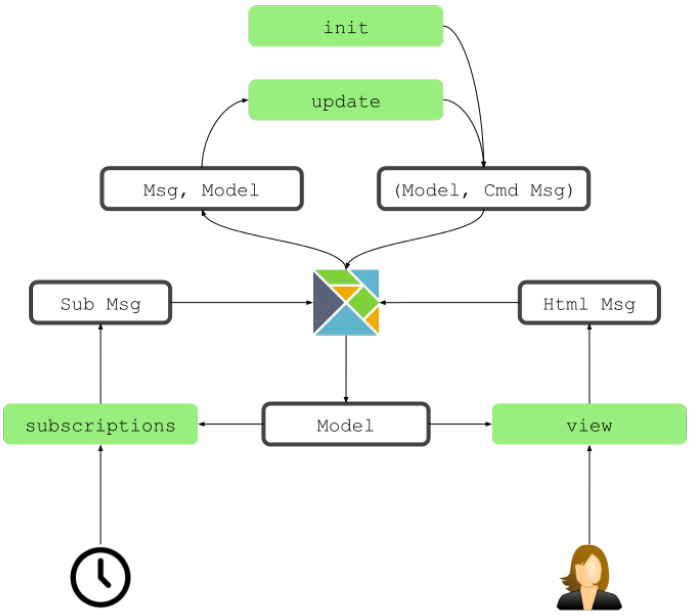


Figure 3.3: The Elm Architecture diagram [56]

3.3 Implementation

The application was implemented in Elm version 0.18. Webpack was used for building and bundling. Using Webpack is a standard for developing web applications. Webpack on its own understands only JavaScript files; for Elm support, I used *elm-webpack-loader* and *elm-hot-loader*. Hot reloading is a technique of watching for changes in source files, recompiling them when needed and then swapping only those that changed. It enables faster and more convenient development. I used Csscomb for auto-formatting CSS files; not so important but still useful, especially if the application would grow in size. All these tools were installed by the Yarn package manager (alternative to NPM) and executed in the Node.js environment.

3.3.1 Deployment diagram

The following diagram shows how the application would be deployed; it also aims to explain how is an Elm application executed in a web browser.

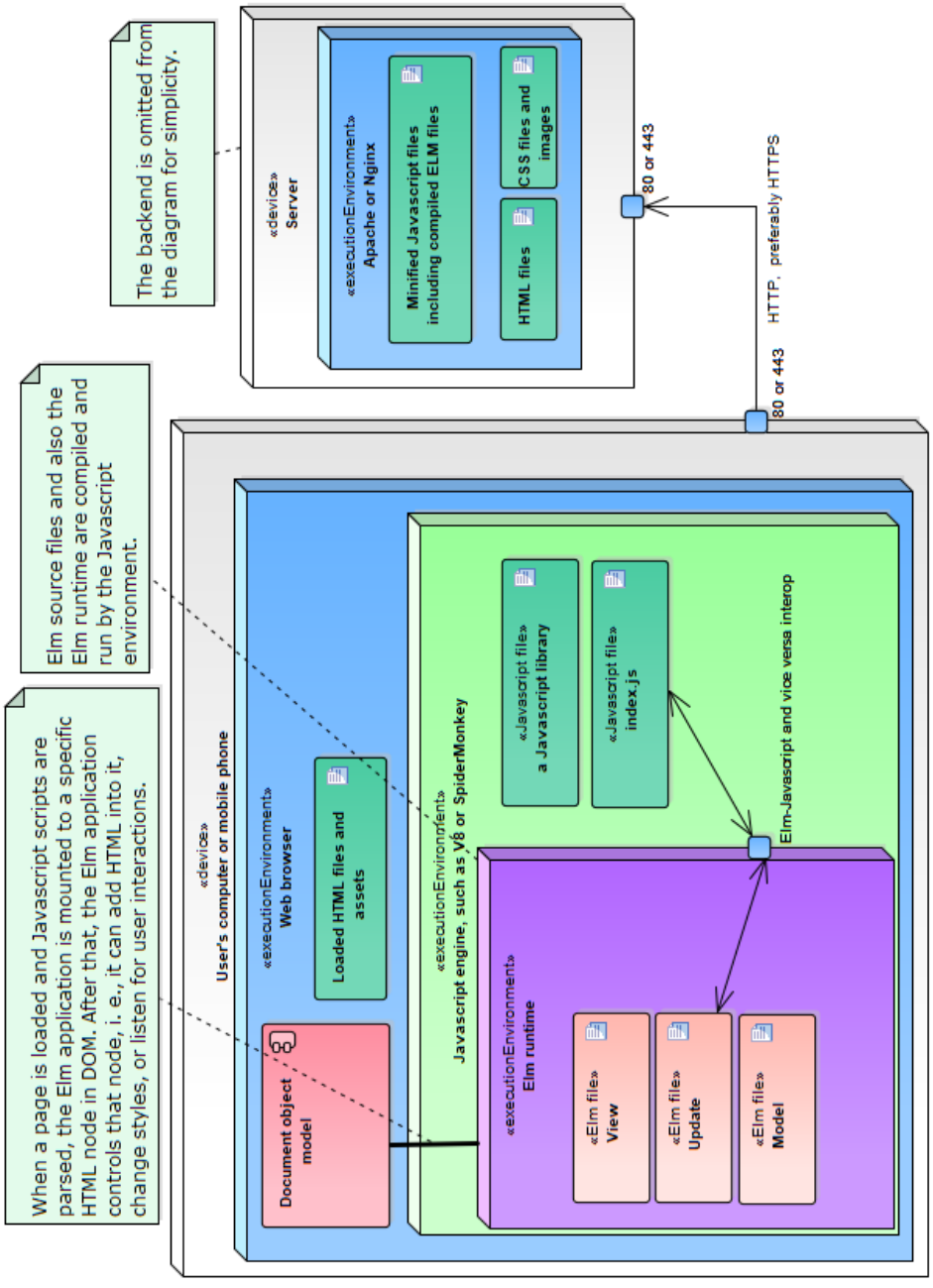


Figure 3.4: Deployment model of an Elm application

3.3.2 Testing

The application is tested with a set of unit tests. I used the *elm-test* package, which is maintained by Elm Community, an *"unofficial group for shared work on Elm packages and documentation"* [57], and which is a usual way of testing Elm applications. It provides basic functionalities for unit testing (asserting, test naming and grouping) and a test runner for Node.js. A test runner for browser exists in an individual package. *Elm-test* also has support for fuzz testing. Fuzz testing is a technique of generating random data of specified type (strings, numbers, booleans) and supplying it as an input for a test. *"If the fuzzer [data generator] can make test fail, it also knows how to shrink that failing input into more minimal examples, some of which might also cause the tests to fail. In this way, fuzzers can usually find the smallest or simplest input that reproduces a bug."* [58] Test examples:

```
1  test "compose modal should be closed after saving message as draft" <|
2    \() ->
3      initialModel
4        |> update ComposeEmail
5        |> Tuple.first
6        |> update (ComposeMsg SaveDraft)
7        |> Tuple.first
8        |> .composedEmail
9        |> Expect.equal Nothing
```

Listing 3.1: Example of Elm unit test

```
1  describe "compose email's"
2  [ fuzz string "body should change according to input" <|
3    \randomString ->
4      initialModel
5        |> update ComposeEmail
6        |> Tuple.first
7        |> update (ComposeMsg (UpdateBody randomString))
8        |> Tuple.first
9        |> .composedEmail
10       |> .body
11       |> Expect.equal randomString
12 ]
```

Listing 3.2: Example of Elm fuzz test

In the second example, the fuzz test is called with random strings approximately one hundred times. Each time it is checked whether the email's body was updated correctly.

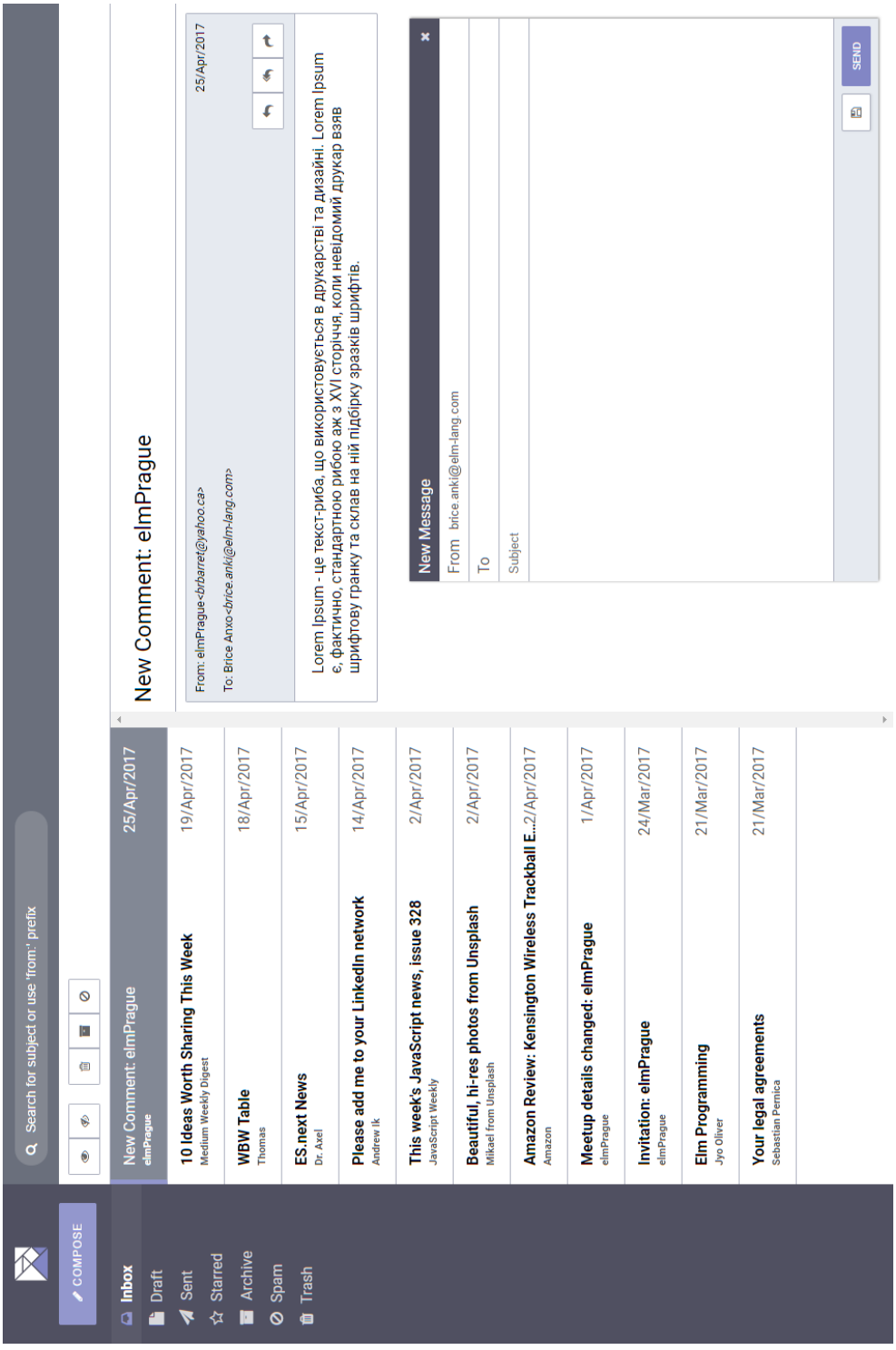


Figure 3.5: Screenshot of the application

Chapter 4

Elm development tools analysis

In this chapter, I discuss differences between JavaScript's and Elm's development tools and libraries.

4.1 Development tools

Development tools are an important part of a programming language (not literally part of the language, but part of the experience working with the language). When a developer, or a team of developers, consider working in a new programming language, quality, maturity, and availability of important tools play a major role in the decision.

4.1.1 Compiler

A compiler is probably the Elm's strongest advantage. Together with a type system, it catches almost all bugs before running in production. It is often said that Elm has no runtime exceptions. Evan explains: *"Unlike hand-written JavaScript, Elm code does not produce runtime exceptions in practice. Instead, Elm uses type inference to detect problems during compilation and give friendly hints. This way problems never make it to your users. NoRedInk [company] has 80k+ lines of Elm, and after more than a year in production, it still has not produced a single runtime exception."* [59] The compiler messages are helpful, always saying what is wrong, underlining the exact position of the problem and offering a solution. I personally also like the "humanity" of the compiler; for example, this is a part of an error message caused by a type mismatch: *"As I infer the type of values flowing through your program, I see a conflict between these two types..."*

JavaScript has no compiler, and therefore, for example, one misspelled property name can crash a whole program. To some extent, a linter can help, but the number of bugs that can a linter find is definitely limited. Another option how to make JavaScript safer is to type check it, see Section 4.2.4.

4.1.2 Elm Reactor

Elm Reactor is several features bundled together: Elm compiler, watching source files for changes, and a local web server. Together it offers interactive development. An Elm application is served on the localhost, change in a source file triggers re-compiling and after refreshing the page, a new version of the application is available. This is known as live reloading.

Elm Reactor has limited features; it does not have functionality for asset loading (CSS, images), bundling, or hot module reloading. It is not suitable for a real-world use. I think the best solution is to use JavaScript's *Webpack*.

4.1.3 Code auto formatter

Another handy tool is *elm-format*; it formats Elm source code according to a standard set of rules based on the official Elm Style Guide. Almost all Elm code I have ever seen was formatted with *elm-format*. Once one get used to it, it helps readability. Recently, a similar tool in the JavaScript ecosystem started to be popular and widely used; it is named *prettier* and it does basically the same for JavaScript as *elm-format* does for Elm.

4.1.4 Package repository

Elm has its own package repository and a command line tool *elm-package* for installing and publishing packages. Source code of packages is hosted on *Github*. Elm's core packages, like *Core*, *Http*, *Html*, are also hosted in the repository along with their documentations. A prominent feature of the repository is that it enforces semantic versioning, a system for increasing package versions in a way that one can safely update dependencies. The repository detects any changes in API of the uploaded package, and if there is a change, it requires an increase of the major version. Versions consist three numbers: major, minor, and patch. Backward-non-compatible changes require increasing the major version.

4.1.5 REPL

Both JavaScript and Elm have read-eval-print-loop. Elm has *elm-repl*, and for JavaScript, one can use either Node.js REPL or services like JSFiddle.

4.1.6 Debugging

As of April 2017, it is not possible to debug Elm code (in the meaning of setting breakpoints and stepping through functions). Elm code is compiled to JavaScript code and then run by a web browser. Technically, a programmer can still use Chrome Development Tools to debug the JavaScript code, but the code is completely different from the original Elm code. Elm, however, has a

different type of debugging that is possible because of the Elm Architecture. In debug mode, every message that passes through the application (through the *update* function) is stored, along with the current state of the application (model). It is then possible to (1) rewind the history – to go back in time, to specific past message and the application state is updated according to the to past state, and (2) to import and export the history of messages. This is called *time travel debugging* and is definitely a handy tool.

I think that the lack of classic debugging in Elm is not a big problem, mainly because every function is pure and can be tested in isolation, and there is no shared state. And from my personal experience, a lot of bugs that I debug in JavaScript, would, if I was working with Elm, be earlier caught be the Elm compiler.

4.1.7 Testing

In Section 3.3.2, I write about testing in Elm; to sum it up, a community package *elm-test* provide functionality for unit testing and it has one extra feature named fuzz testing, which most JavaScript test frameworks do not support. Fuzz testing is taking advantage of the purity of Elm functions. *elm-test* or any other Elm test package do not support test coverage, a measure used to describe the degree to which the source code of a program is executed when tests are run.

4.2 Libraries

There is an important group of libraries that exist in JavaScript and are hugely used, but are not present in Elm. Not because nobody created them, but because their functionality is incorporated in the Elm language itself, and they are therefore not needed. View rendering, state management, immutability, type checking; these are functionalities that in JavaScript are provided by multiple libraries, but in Elm, they are just in the language. I think that the JavaScript's diversity brings benefits (probably mostly for the web development community as a whole), but also that the Elm's approach of having one clear way of doing things can increase productivity and free programmer's mental resources for other and more complicated matters.

4.2.1 View rendering

Rendering the view is an important part of developing a web application. In JavaScript, there is a multitude of libraries (React, Vue, Cycle.js, ...), each with its own approach, philosophy, and API. A view written in one library is not compatible with other libraries and developers need specific knowledge of frameworks. On the other hand, view rendering in Elm is just a couple of functions, pure functions as everything else.

4.2.2 State management

The situation with state management is the same as with view rendering. These are some examples of how can the state be handled in JavaScript: Flux, Redux (Redux is inspired by Elm's state management), MobX, local state, global state. Elm offers one recipe – the Elm Architecture.

4.2.3 Immutability

A widely used JavaScript library for immutability is Facebook's Immutable.js. It provides many persistent immutable data structures including List, Stack, Map, OrderedMap, Set, OrderedSet, and Record. One caveat is that using immutable and mutable structures together in one project may confuse developers and make them remember, or constantly check, what type of structure they are just working with. In Elm, the immutability of data structures is built in the language and is not optional.

4.2.4 Type checking

Elm is statically typed (for a review of Elm's type system see Section 2.6.2). In JavaScript, there are two major ways how to introduce types. (1) With a static type checker, such as Facebook's Flow, a programmer can type-annotate his JavaScript code. A type checker then infers types, validate them, and remove type annotations to produce valid JavaScript code (code with type annotations is invalid for an interpreter). (2) Use TypeScript, a typed superset of JavaScript. Both these solutions share that their type system is fault tolerant (TypeScript with default settings outputs JavaScript code even in presence of type errors) and that they support gradually typing of JavaScript code (both Flow and Typescript have type *any* that basically turns off type checking for values annotated by it). TypeScript from version 2.3 defaults to strict type checking, i.e., no implicit *any* type and strict null checks.

4.2.5 Other more specialized libraries

So far I discussed libraries that are useful for almost any project. Regarding more specialized libraries (graph visualization, UI kits, databases), I think it is safe to say that JavaScript has much more of these, which also have more features and are better maintained. Simply because there are many more JavaScript developers, who had more time (JavaScript is over twenty years old), and also there is a lot of companies that have they businesses build around JavaScript (especially Node.js, React, Vue, Angular). In contrast to Elm that has fewer developers and around three years of existence. Elm is also an evolving language, for example changes from version 0.16 to 0.17 (current is 0.18) were breaking changes, backward-non-compatible.

4.3 Summary

In my opinion, the Elm's approach of having one clear way of doing things and the fact that some important functionalities are incorporated in the language, together make a better developer experience, lower complexity, and lead to an increased productivity. Regarding the productivity, I am speaking from my experience of working on the prototype Inbox application and my experience with JavaScript web development.

To sum up, a compiler is the Elm's biggest strength and its tooling is sufficient. Elm has fewer libraries than JavaScript, but some important ones are built in the Elm language.

Conclusion

Important concepts of functional programming had been reviewed, which together with the introduction of programming languages Haskell and JavaScript, laid a basis for an inspection of the Elm language. The inspection had three levels. (1) A theoretical level in Review Chapter focused on the syntax, the type system, data structures, and on the Elm Architecture. (2) A practical level in Application in Elm Chapter focused on testing, deployment, and on a software development process. (3) In Elm development tools analysis Chapter focused on the compiler, packages, debugging, and also on development tools and available Elm libraries.

The first research question, *"How does Elm implement the key concepts of functional programming?"*, was answered in Elm language Section 2.6. The key Elm's functional properties are that all functions are pure and curried, all data structures are immutable, and side effects are managed by the runtime. The answer to the second research question, *"Are Elm and its development tools ready for real-world use?"*, was elaborated in Chapter 4. My conclusion is that Elm is ready for real-world use.

This thesis together with application's source codes might be a useful study material for the Elm language and I plan to share it with Elm developer communities.

Personal growth

I have learnt a lot by writing this thesis. I have got to know a delightful programming language and I have hugely improved in written English.

Bibliography

1. *Overview of the Four Main Programming Paradigms* [online] [visited on 2017-02-11]. Available from: http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html#paradigms_imperative-paradigm-overview_title_1.
2. HANUS, Michael. Multi-Paradigm Declarative Languages. In: *Logic Programming* [online]. Berlin: Springer Berlin Heidelberg, 2007, vol. 4670, pp. 45–75 [visited on 2017-02-23]. ISBN 978-3-540-74608-9. Available from DOI: 10.1007/978-3-540-74610-2_5.
3. TORGERSSON, Olof. A Note on Declarative Programming Paradigms and the Future of Definitional Programming. *Das Winteroete* [online]. 1996, vol. 96, no. 1996, pp. 13 [visited on 2017-02-23]. Available from: http://www.cse.chalmers.se/~oloft/Papers/wm96/wm96.html.
4. ERIKSSON, Nils; ÄRLERYD, Christofer. *Elmulating JavaScript* [online]. 2016 [visited on 2017-02-23]. Available from: http://www.diva-portal.org/smash/record.jsf?pid=diva2:954012.
5. CZAPLICKI, Evan. *Elm: Concurrent FRP for Functional GUIs*. Harvard University, 2012. Available also from: http://www.testblogpleaseignore.com/wp-content/uploads/2012/03/thesis.pdf.
6. *Declarative Programming* [online] [visited on 2017-02-11]. Available from: http://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html#detail.
7. HUGHES, J. Why Functional Programming Matters. *The Computer Journal* [online]. 1989, vol. 32, no. 2, pp. 98–107 [visited on 2017-02-23]. ISSN 0010-4620. Available from DOI: 10.1093/comjnl/32.2.98.
8. MICHAELSON, Greg. *An Introduction to Functional Programming through Lambda Calculus*. Dover ed. Mineola, N.Y: Dover Publications, 2011. Dover books on mathematics. ISBN 978-0-486-47883-8. OCLC: ocn630478012.

9. CARL, Burch. *Lambda Calculus* [online] [visited on 2017-02-24]. Available from: http://www.toves.org/books/lambda/.

10. Lazy Evaluation. In: *HaskellWiki* [online] [visited on 2017-02-24]. Available from: https://wiki.haskell.org/Lazy_evaluation.

11. *HaskellWiki* [online] [visited on 2017-02-24]. Available from: https://wiki.haskell.org/Haskell.

12. *Java Documentation - Java.Util.Stream* [online] [visited on 2017-02-26]. Available from: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

13. *Cpp Reference* [online] [visited on 2017-02-26]. Available from: http://en.cppreference.com/w/cpp/algorithm.

14. *PHP: Array Functions - Manual* [online] [visited on 2017-02-26]. Available from: http://php.net/manual/en/ref.array.php.

15. *Python 3.6.0 Documentation* [online] [visited on 2017-02-26]. Available from: https://docs.python.org/3.6/library/functions.html.

16. *Mapping and Filtering* [online] [visited on 2017-02-26]. Available from: http://people.cs.aau.dk/~normark/prog3-03/html/notes/higher-order-fu_themes-map-filter-section.html.

17. Fold. In: *HaskellWiki* [online] [visited on 2017-02-26]. Available from: https://wiki.haskell.org/Fold.

18. DILLER, Antoni. *Haskell Unit 6: The Higher-Order Fold Functions* [online] [visited on 2017-02-26]. Available from: http://www.cantab.net/users/antoni.diller/haskell/units/unit06.html.

19. Currying. In: *HaskellWiki* [online] [visited on 2017-02-26]. Available from: https://wiki.haskell.org/Currying.

20. Partial Application. In: *HaskellWiki* [online] [visited on 2017-02-26]. Available from: https://wiki.haskell.org/Partial_application.

21. Side Effect. In: *Wikipedia* [online]. 2017 [visited on 2017-02-26]. Available from: https://en.wikipedia.org/w/index.php?title=Side_effect_(computer_science)&oldid=760895851.

22. *Redux* [online] [visited on 2017-03-05]. Available from: http://redux.js.org/docs/introduction/CoreConcepts.html.

23. Persistent Data Structure. In: *Wikipedia* [online]. 2016 [visited on 2017-03-04]. Available from: https://en.wikipedia.org/w/index.php?title=Persistent_data_structure&oldid=734240973.

24. Lisp (Programming Language). In: *Wikipedia* [online]. 2017 [visited on 2017-03-05]. Available from: https://en.wikipedia.org/w/index.php?title=Lisp_(programming_language)&oldid=767986041.

25. Hindley–Milner Type System. In: *Wikipedia* [online]. 2017 [visited on 2017-03-05]. Available from: https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system.

26. *Haskell 2010 Language Report* [online] [visited on 2017-03-09]. Available from: https://www.haskell.org/onlinereport/haskell2010/.

27. *Haskell: Type Classes* [online] [visited on 2017-04-28]. Available from: https://www.haskell.org/tutorial/classes.html.

28. All About Monads. In: *HaskellWiki* [online] [visited on 2017-03-10]. Available from: https://wiki.haskell.org/All_About_Monads.

29. IO Monad. In: *HaskellWiki* [online] [visited on 2017-03-11]. Available from: https://wiki.haskell.org/All_About_Monads#The_IO_monad.

30. Monad. In: *Wikipedia* [online]. 2017 [visited on 2017-03-10]. Available from: https://en.wikipedia.org/w/index.php?title=Monad_(functional_programming)&oldid=767819858.

31. Monad. In: *HaskellWiki* [online] [visited on 2017-03-11]. Available from: https://wiki.haskell.org/Monad.

32. *Pattern Matching* [online] [visited on 2017-03-09]. Available from: https://www.haskell.org/tutorial/patterns.html.

33. Algebraic Data Type. In: *Wikipedia* [online]. 2017 [visited on 2017-03-10]. Available from: https://en.wikipedia.org/w/index.php?title=Algebraic_data_type&oldid=763558354.

34. *Types in Haskell* [online] [visited on 2017-03-10]. Available from: https://www.haskell.org/tutorial/goodies.html.

35. Algebraic Data Type. In: *HaskellWiki* [online] [visited on 2017-03-10]. Available from: https://wiki.haskell.org/Algebraic_data_type.

36. FLANAGAN, David. *JavaScript: The Definitive Guide*. Fifth edition. O'Reilly, 2006. ISBN 978-0-596-10199-2.

37. JavaScript Engine. In: *Wikipedia* [online]. 2017 [visited on 2017-03-11]. Available from: https://en.wikipedia.org/w/index.php?title=JavaScript_engine&oldid=762516278.

38. *TC39 - ECMAScript* [online] [visited on 2017-03-11]. Available from: https://www.ecma-international.org/memento/TC39-M.htm.

39. RAUSCHMAYER, Axel. *Exploring ES6* [online]. Leanpub, 2015 [visited on 2017-03-11]. Available from: https://leanpub.com/exploring-es6.

40. *Immutable.Js* [online] [visited on 2017-03-11]. Available from: https://facebook.github.io/immutable-js/.

41. *Clojure* [online] [visited on 2017-03-11]. Available from: https://clojure.org/.

42. ANTUKH, Andrey; GÓMEZ, Alejandro. *ClojureScript Unraveled* [online] [visited on 2017-02-11]. Available from: `https://funcool.github.io/clojurescript-unraveled/`.

43. *Differences from Clojure* [online] [visited on 2017-03-11]. Available from: `https://clojurescript.org/about/differences`.

44. *PureScript* [online] [visited on 2017-04-28]. Available from: `http://www.purescript.org/`.

45. BLACKHEATH, Stephen; JONES, Anthony. *Functional Reactive Programming*. Shelter Island, NY: Manning Publications Co, 2016. ISBN 978-1-63343-010-5. OCLC: ocn907203779.

46. *Terminology - What Is (Functional) Reactive Programming?* [online] [visited on 2017-03-12]. Available from: `http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming`.

47. CZAPLICKI, Evan. *An Introduction to Elm* [online] [visited on 2017-04-01]. Available from: `https://guide.elm-lang.org/`.

48. Type System. In: *Wikipedia* [online]. 2017 [visited on 2017-03-17]. Available from: `https://en.wikipedia.org/w/index.php?title=Type_system&oldid=769381407`.

49. *Elm Package - Maybe* [online] [visited on 2017-03-30]. Available from: `http://package.elm-lang.org/packages/elm-lang/core/latest/Maybe`.

50. POUDEL, Pawan. *Elm Programming* [online] [visited on 2017-03-17]. Available from: `http://elmprogramming.com`.

51. *Elm Package - Basics* [online] [visited on 2017-03-28]. Available from: `http://package.elm-lang.org/packages/elm-lang/core/2.1.0/Basics`.

52. *Elm Package - Core* [online] [visited on 2017-03-30]. Available from: `http://package.elm-lang.org/packages/elm-lang/core/latest/`.

53. *Elm Package - Tuple* [online] [visited on 2017-03-30]. Available from: `http://package.elm-lang.org/packages/elm-lang/core/5.1.1/Tuple`.

54. *Elm Guide - Type Alias* [online] [visited on 2017-03-30]. Available from: `https://guide.elm-lang.org/types/type_aliases.html`.

55. CZAPLICKI, Evan. *Blazing Fast Html Round Two* [online] [visited on 2017-04-01]. Available from: `http://elm-lang.org/blog/blazing-fast-html-round-two`.

56. *Elm-Lifecycle - A Symmetrical Simplified Elm Lifecycle Diagram* [online] [visited on 2017-04-16]. Available from: `https://github.com/plaxdan/elm-lifecycle`.

57. *Github - Elm-Community/Elm-Test* [online] [visited on 2017-04-20]. Available from: `https://github.com/elm-community/elm-test`.

58. *Elm Package - Fuzz* [online] [visited on 2017-04-20]. Available from: `http://package.elm-lang.org/packages/elm-community/elm-test/3.1.0/Fuzz`.

59. *Elm Lang Page* [online] [visited on 2017-04-27]. Available from: `http://elm-lang.org/`.

Appendix A

Acronyms

API Application Programming Interface

CSS Cascading Style Sheets

DOM Document Object Model

FRP Functional Reactive Programming

GUI Graphical user interface

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

REPL Read–eval–print loop

SPA Single-page application

SQL Structured Query Language

UML Unified Modeling Language

Appendix B

Contents of enclosed USB drive

- `readme.txt` ..... the file with USB drive contents description
- `app`
  - `normal` ..... the directory of the compiled application
  - `debug` ..... the directory of the compiled application in debug mode
- `src` ..... the directory of source codes
  - `app` ..... the directory of source codes of the application
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format