**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**

# THESIS TITLE

**NÁZEV PRÁCE**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**

**AUTHOR** **JAKUB ZÁRYBNICKÝ**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. ONDŘEJ LENGÁL, Ph.D.**
**VEDOUCÍ PRÁCE**

**BRNO 2019**

## Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.



## Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.



## Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

## Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.


## Reference

ZÁRYBNICKÝ, Jakub. *Thesis title.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

# Rozšířený abstrakt

Do tohoto odstavce bude zapsán rozšířený abstrakt práce v českém jazyce, bude mít rozsah 2 až 6 normostran a bude obsahovat úvod, popis vlastního řešení a shrnutí a zhodnocení dosažených výsledků.

# Thesis title

## Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. X. The supplementary information was provided by Mr. Y. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . . . . . . . . . . . . . . . . . . . . .
Jakub Zárybnický
March 14, 2019

# Contents

# List of Figures

# List of Listings

# Chapter 1

# Introduction

Imagine you are building an application for an event your company is organizing. There will not be reliable internet connection in the area of the event, so it needs to work offline. You are on a tight budget, so implementing one version for each platform is not feasible, but it needs to work reliably across all mobile platforms and ideally in the browser as well. What's the easiest way to accomplish that?

This situation is exactly where a rather new concept of a 'Progressive Web Application' (PWA) is the best solution. Generally said, a PWA is website capable of running offline, using mobile notifications, or synchronizing data in the background - things previously specific to native mobile applications.

Now additionally consider that the application server is written in Haskell, a statically typed, purely functional programming language. We want to reuse the business logic already written there to avoid duplicating code, so we search for a way to write a Web application in Haskell. We find a find many resources and a quickly growing community but while creating the application, we soon step into the unknown. A medium-scale application needs a large number of capabilities, but the ecosystem of frontend Haskell is not yet big enough to support many of them - most libraries in the area are either exploratory, or one-off projects.

In this work, we will try to fill in many such gaps for Haskell developers of frontend applications, with a specific focus on Progressive Web Applications. We will first go through what specifically is missing in the Haskell ecosystem (Section 2) and implement the missing pieces by hand in three separate applications (Section 3). From those applications, we will then design and extract a set of reusable components (Section 4), and then look [TODO: fill in after the practicals, guidelines or recommendations].

## 1.1 Related work

The volume of work in area of frontend Haskell is not large, as the Haskell-to-JavaScript compiler, GHCJS, is only available since 2013, also due to the fact that Haskell in general is only recently being accepted as a mainstream language. Academic work in this area is sparse, but there are several mature projects - mainly commercially sponsored ones - under active development. Reflex and Obelisk are a UI framework and a deployment tool respectively, from Obsidian Systems [35]. Tweag [41] is working on a Haskell-to-WebAssembly compiler, Asterius, and QFPL [21] has created many learning materials for frontend Haskell.

# Chapter 2

# Technologies

We will first have a more detailed look at the technologies mentioned above - at what the term Progressive Web Application means and why would one want to use Haskell for creating browser applications.

## 2.1 Progressive Web Application

The term 'Progressive Web Application' is an umbrella term for several relatively new, closely related technologies. It continues in the general trend of expanding the capabilities of browser applications and of closing the gap between browser and native mobile applications. While many of these technologies are useful also on desktop, the main target audience are mobile Web browsers.

The term Progressive Web Applications has an exact specification in a checklist created by [15], which describes two levels of PWAs, a 'Baseline PWA' and an 'Exemplary PWA'. The defining characteristics of a Baseline PWA are:

- Pages are responsive on tablets & mobile devices
- All app URLs load while offline
- Metadata provided for Add to Home screen
- Page transitions do not feel like they block on the network
- Each page has a URL
- Pages use the History API
- Site uses cache-first networking
- Site appropriately informs the user when they are offline
- Push notifications (consists of several related requirements)

While there are several more requirements for an Exemplary PWA, it is just the baseline ones that we will focus on. The technologies used to fulfill these requirements are relatively recent developments, but they are supported in all the major Web browsers. They are:

- Web App Manifest - a specification for the centralized location of application metadata - its name, icons, display mode, ...
- Service Workers - 'a scriptable network proxy' [28]. The technology that takes care of offline-availability, push notifications, and background synchronization.
- IndexedDB - a storage location that is accessible from the browser as well as the service worker, it allows background sync to work with the application state directly
- Web Platform APIs - a set of APIs that expose capabilities of the underlying system - examples include geolocation or audio/video capture [2]

Some notable examples of companies using PWAs are: Twitter [43], Uber [44], or Flipkart [13].

## 2.2 Haskell

```haskell
type HackageAPI =
  "users" :> Get '[JSON] [User] :<|>
  "user" :> Capture "login" Login :> Get '[JSON] User :<|>
  "packages" :> Get '[JSON] [Package]

getUsers :: Handler [User]
getUser :: Login -> Handler User
getPackages :: Handler [Package]

server :: Server HackageApi
server = getUsers :<|> getUser :<|> getPackages

getUsers :<|> getUser :<|> getPackages =
  client @HackageApi "http://hackage.haskell.org"
```

Listing 1: An example of a web server in Haskell

Haskell is described as a "statically typed, purely functional programming language with type inference and lazy evaluation" [20]. It is originally a research language, developed as a vehicle for new research in the area of programming languages since 1990 [17]. It has served as such, and in fact it still is the target of active research - some more prominent projects are Dependent Haskell [10] and Linear Haskell [3].

Only recently has it been used in commercial work, as exemplified by Facebook's Haskell spam filter [23]. While there are many benefits to using a strongly typed functional language - it eliminates entire classes of programming errors [30], anecdotally shown by the common saying that "If it compiles, it works" - it is conceptually different from languages commonly taught at universities.

As for using Haskell in the browser, it may seem strange at first glance to want such a thing when JavaScript is the only language supported by Web browsers. There is however a growing number of languages that compile to JavaScript, that use it as their compile target instead of Assembly or LLVM, which can be done either by translating the logic of the program into JavaScript as is (transpiling), or by implementing an alternative runtime environment in JavaScript which then interprets the byte- or source-code. Another technology that enables languages to run in the browser is WebAssembly, an alternative assembly language and a runtime designed specifically for the Web.

Web developers have been using JavaScript compilers for a long time - CoffeeScript is rather popular language announced in 2010 [1]. Also the new ECMAScript 6 or 7 features have only been usable via 'transpilers' until browsers implemented them natively, transpilers like Babel [25]. There are other, more advanced languages build with compilation to JavaScript in mind, e.g. TypeScript, a superset of ECMAScript 6 [27], or Elm, a framework with its own language based on Haskell [7]. The need to compile your code before running it is now quite accepted in the world of Web development.

The currently accepted way of running Haskell in the browser is via GHCJS, a Haskell-to-JavaScript compiler, although there are two active projects in the process of creating a Haskell-to-WebAssembly compiler - WebGHC [12] and Asterius [42].

## 2.3 Nix

```
{ stdenv, fetchurl, perl }:

stdenv.mkDerivation {
  name = "hello-2.1.1";
  builder = ./builder.sh;
  src = fetchurl {
    url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
    sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7w1800any6465";
  };
  inherit perl;
}
```

Listing 2: An example Nix derivation of GNU hello

One technology not yet mentioned, but upon which will stand the entire build system used in this work from compiling to deploying, is Nix. Nix [8] is a package manager with focus on reproducibility and isolation, described as a 'purely functional package manager' - every package is built in isolation by a 'pure function' without side-effects, and the result is immutable. When installing a package packaged in such a way, the exact versions of dependencies are installed and used as well (identified by a SHA-256 hash) - all runtime dependencies all the way up to `libc`.

Nix is a declarative build tool and a package manager, similar in purpose to Make and in philosophy to Haskell. There are other tools built on top of Nix though, the most interesting being NixOS, a declarative operating system, and NixOps, a cloud deployment tool [9]. Nix shines at cross-compilation, which is the main reason we will use it in this work - compiling to JavaScript or Android/iOS is trivial after the initial setup.

# Chapter 3

# Research

As this work does not live in a vacuum, we also need to consider commonly used Web frameworks and platforms and decide what features to implement in. We will first walk through the features that frameworks today implement, describe them and define the relevant terms. Afterwards, we will have a brief look at the specifics of the JavaScript ecosystem - the most common language in Web development - and the of Haskell, my language of choice, and try to find the places where Haskell is lagging behind and especially the features that we will need to fill in in this work.

## 3.1 Features of Web frameworks

The basis of a web framework is the **UI toolkit**, which defines the structure, architecture and paradigm of the rest of the application. I am intentionally using the now-uncommon term 'toolkit', as the UI frameworks we will see vary in their scope - e.g. React is just a library with a small API, whereas Angular provides a quite opinionated platform. Individual frameworks are quite disparate, with large differences in the size of their community, maturity, developer friendliness and the breadth of features or available libraries.

Frameworks usually have one defining feature they are built around (virtual DOM for React or event streams for Angular), but there are many other concerns that a framework needs to take care of. **Templating** is one of the essential ones. It is a way of composing the HTML that makes up an application which also usually includes some 'view logic' and variable interpolation. In some frameworks the whole program is a template (purely functional React), some have templates in separate files and pre-compile them during the build process or even in the browser (Angular). Templates may also contain CSS as well - see the new CSS-in-JS trend.

The second defining feature of frameworks is **state management**. This rather vague concept may include receiving input from the user, displaying the state back to the user, communicating with APIs and caching the responses, etc. While state management is simple at a small scale, there are many problems that appear only in larger applications with several developers. Some approaches include: a 'single source of the truth' and immutable data (Redux), local state in hierarchical components (Angular), or unidirectional data flow with several entity stores (Flux).

Another must-have feature of a framework is **routing**, which means manipulating the displayed URL using the History API, and changing it to reflect the application state and vice-versa. It also includes switching the application to the correct state on start-up. While the router is usually a rather small component, it is fundamental to the application in the same way the previous two items are.

A component where frameworks differ a lot is a **forms** system. There are a few layers of abstraction at which a framework can decide to implement forms, starting at raw DOM manipulation, going on to data containers with validation but manual rendering, all the way up to form builders using domain-specific languages. The topic of 'forms' includes rendering a form and its data, accepting data from the user and validating it, and sometimes even submitting it to an API.

There are other features that a framework can provide - authentication, standardized UI components, and others - but frameworks usually leave these to third party libraries. There is one more topic I would like to mention that is usually too broad to cover in the core of a framework, but important to consider when developing an application. **Accessibility** is an area concerned with removing barriers that would prevent any user from using a website. It has many parts to it - while the focus is making websites accessible to screen-readers, it also includes supporting other modes of interaction, like keyboard-only interaction. Shortening **load times** on slow connections also makes a website accessible in parts of the world with slower Internet connections, and supporting **internationalization** removes language and cultural barriers.

Accessibility is something that requires framework support on several levels. Making a site accessible requires considerations during both design (e.g. high color contrast) and implementation (semantic elements and ARIA attributes), and that is usually left up to application code and accessibility checklists, with the exception of some specialized components like keyboard focus managers. There are however tools like aXe-core that check how accessible a finished framework is, and these can be integrated into the build process.

**Internationalization** is somewhat easier to support in a framework, as it does include so many cross-cutting concerns. At the most basic level, it means simple string translations, perhaps with pluralization and word order. Going further, it may also mean supporting RTL scripts, different date/time formats, currency, or time zones.

As for **load times**, there are many techniques frameworks use to speed up the initial load of an application. We can talk about the first load, which can be sped up by compressing assets (CSS, fonts, fonts or scripts) and removing redundant ones, or by preparing some HTML that can be displayed to the user while the rest of the application is loading to increase the perceived speed. After the first load, the browser has some of the application's assets cached, so loading will be faster. One of the requirements of a PWA is using the Service Worker for instantaneous loading after the first load.

There are two patterns of preparing the HTML that is shown while the rest of the application is loading - so called **prerendering**. One is called 'app shell', which is a simple static HTML file that contains the basic structure of the application's layout. The other is 'server-side rendering', and it is a somewhat more advanced technique where the entire contents of the requested URI is rendered on the server including the data of the first page, and the browser part of the application takes over only afterwards, without the need to fetch any more data. There is another variant of 'server-side rendering' called the 'JAM stack' pattern [39], where after application state changes, the HTML of the entire application, of all application URLs is rendered all at once and saved so that the server does not need to render the HTML for every request. These techniques are usually part of a framework's **supporting tools**, about which we will talk now.

Developers from different ecosystems have wildly varying expectations on their tools. A Python developer might expect just a text editor and an interpreter, whereas a JVM or .NET developer might not be satisfied with anything less than a full-featured IDE. We will start with the essentials, with **build tools**. Nowadays, even the simplest JavaScript application usually uses a build step that packages all its source code and styles into a single bundle for faster loading. A framework's tool-chain may range from a set of conventions on how to use the compiler that might get formalized in a Makefile, through a CLI tool that takes care of building, testing and perhaps even deploying the application, to the way of the IDE, where any build variant is just a few clicks away.

**Debugging tools** are the next area. After building an application, trying it out, and finding an error, these tools help in finding the error. There are generic language-specific tools - a stepping debugger is a typical example - and there are also framework-specific tools, like an explorer of the component hierarchy (React) or a time-traveling debugger (Redux). In the web world, all modern browsers provide basic debugging tools inside the 'DevTools' - a stepping debugger and a profiler. Some frameworks build on that and provide an extension to DevTools that interacts with the application in the current window, some provide debugging tools integrated into the application itself.

When building or maintaining a large application with several developers, it is necessary to ensure good practices in all steps of the development process. There are two general categories in **quality assurance** tools - testing (dynamic analysis) tools and static analysis tools. In the commonly used variants, tests are used either as an aid while writing code (test-driven development), or to prevent regressions in functionality (continuous integration using unit tests and end-to-end tests). Static analysis tools are, in the general practice, used to ensure a consistent code style and prevent some categories of errors ('linters'). Frameworks commonly provide pre-configured sets of tools of both types. If necessary - e.g. in integration testing where the burden of set up is bigger - they also provide utility libraries to ease the initial set up. Some frameworks also use uncommon types of tests like 'marble tests' used in functional reactive programming systems.

**Editor integration** is also important in some ecosystems. This includes common features of Integrated Development Environments like auto-completion or refactoring tools. Recently the Language Server Protocol (LSP) [26] project played a big role in allowing editors to support a wide variety of languages by implementing just an LSP client and being able to communicate with any language-specific language server. There are some parts of editor support that can be framework-specific like supporting an embedded domain-specific language or integrating framework-specific debugging tools.

While we were talking about Web frameworks so far, some of them support not only running inside the browser but also being packaged as a **mobile app** for Android or iOS, or as a **native desktop application** for the many desktop operating systems. For mobile support, frameworks often provide wrappers around Apache Cordova, which is a thin wrapper around a regular website exposing some extra capabilities of the device. Some, however, go even further and support fully native mobile interfaces controlled by JavaScript, like React Native. The situation is similar for desktop support, just with Electron used as the base instead of Cordova. The main benefits of packaging a Web application instead just running it inside a browser are performance (they are usually faster to load and to use), access to device-specific capabilities (direct access to the file system), or branding.

The last point in this section is **code generators**. of which there are two variants: project skeleton generators, which create all files necessary for a project to compile and run, and which are provided in a large majority of frameworks. Then there are component generators, which may include generating a template, a URL route and its corresponding controller, or an entire subsection of a website. These are less common but some frameworks also provide them.

## 3.2 JavaScript ecosystem

Moving on, we will take a quick tour of the JavaScript ecosystem and what the library ecosystem looks there, following the same general structure as we have used in the section above.

The most popular **UI toolkits** in JavaScript are currently Angular [14] and React [11]. Vue.js [45] is another, a relatively new but quickly growing one. Of these, Angular is the framework closest to traditional frameworks where it tries to provide everything you might need to create an application. React and Vue are both rather small libraries but with many supporting tools and libraries that together also create a platform, although they are much less cohesive than Angular's platform.

There are fundamental architectural differences between them. Angular uses plain HTML as a base for its templates, and uses explicit event stream manipulation for its data flow. React uses a functional approach where a component is (de facto) just a function producing a JavaScript object, in combination with an event-driven data flow. Vue uses HTML, CSS and JavaScript separately for its templates, and its data flow is a built-in reactive engine.

The most common complaint about the JavaScript ecosystem in general is that it is a 'jungle'. There are dozens or hundreds of small libraries doing the same thing, most however incomplete or unmaintained, with no good way to decide between them. Frameworks avoid this problem by having a recommended set of libraries for common use cases. A different but related complaint is called the 'JavaScript fatigue'. The trends change quickly in the JavaScript ecosystem, libraries come and go each year, a common belief is that if you are not learning at least one new framework per year, you are missing out on opportunities.

As for the individual frameworks mentioned above: Angular is an integrated framework that covers many common use cases in the basic platform. To some though, it is too opinionated, too complex to learn easily, or with too much abstraction to understand.

React and Vue are rather small libraries which means they are very flexible and customizable. There are many variants of libraries for each feature a web application might need, which also means that it is easy to get stuck deciding on which library to pick out of the many options. There are React and Vue 'distributions', however, that try to avoid this by picking a set of libraries and build tools that works together well.

As for the topics mentioned in the previous section - routing, forms, build tools, mobile and desktop applications - most are built into Angular, and for React and Vue there are dozens of options of third party libraries. In my investigation, I have not found a weak side to any of them - which is just what I expected, given that JavaScript is the native language of the Web.

## 3.3 Haskell ecosystem

Going on to the Haskell ecosystem, we will also walk through it using the structure from the 'Features' section. There is significant focus on the semantics of libraries in the Haskell community, e.g. writing down mathematical laws for the foundational types of a library and using them to prove correctness of the code, so UI libraries have mostly used Functional Reactive Programming (FRP) or its derivatives like 'the Elm architecture' [22] as their basis, as traditional imperative event-based programming does not fit those criteria well.

There are five production-ready UI toolkits for the Web that I have found. Of these five, React-flux and Transient are unmaintained, and Reflex, Miso, and Concur are actively developed and ready for production use. Each one uses a conceptually different approach to the problem of browser user interfaces, and they differ in their maturity and the size of their community as well.

**Reflex** [31] (and Reflex-DOM [32], its DOM bindings) looks like the most actively maintained and developed one. Reflex is also sponsored by Obsidian Systems [35] and is the most popular frontend framework in the Haskell community, so its future seems promising. Reflex follows the traditional FRP approach with events and behaviors (adding 'dynamics'), and building a rich combinator library on top of them.

```
main :: IO ()
main = mainWidget $ display =<< count =<< button "Click me"
```

Listing 3: An example of Reflex code (a counter)

**Miso** [19] is a re-implementation of the 'Elm architecture' in Haskell, which means that is uses strictly uni-directional data-flow with a central data store on the one side, and the view as a pure function that takes the state and creates a view on the other, where the view can change the state using strictly defined events. The ecosystem of Miso is not as well developed as Reflex's, and the overall architecture is very limiting - which I consider a large disadvantage.

**Concur** [18] tries to explore a different paradigm by combining 'the best of' the previous two approaches. The developers have so far been focusing on exploring how this paradigm fits into browser, desktop or terminal applications, so it has a quite small range of features. It is a technology I intend to explore in the future when it is more mature, which however does not seem suitable for a large application so far, at least compared to its competitors.

In all of these frameworks, **templating** is a feature that has been side-stepped by creating a domain-specific language for HTML mixed with control flow. There have been attempts at creating a more HTML-like language embedded into Haskell or external templates, though there is no such project that is both feature-complete and actively maintained. It is however possible to reuse existing JavaScript components using the foreign function interface (FFI) between Haskell and JavaScript, and that it exactly what one of the unmaintained frameworks did to use React as its backend (react-flux).

```haskell
type Model = Int

data Action = AddOne
  deriving Eq

main :: IO ()
main = JSaddle.run 8080 $ startApp App {..}
  where
    initialAction = AddOne
    model  = 0
    subs   = []
    events = defaultEvents
    mountPoint = Nothing

    update AddOne m = noEff (m + 1)

    view x = div_ []
      [ text (ms x)
      , button_ [ onClick AddOne ] [ text "Click Me" ]
      ]
```

Listing 4: An example of Miso code (a counter)

```haskell
main :: IO ()
main = do
  initConcur
  void $ runWidgetInBody $ void $ flip execStateT (0 :: Int) $
    forever $ increment1 <|> displayCount
  where
    increment1 = lift (el_ E.div [] $ button "Click Me") >> modify (+10)
    displayCount = do
      count <- get
      lift $ el_ E.div [] $ text $ show count ++ " clicks"
```

Listing 5: An example of Concur code (a counter)

**State management** is where the frameworks differ the most. Miso follows the Elm architecture strictly with a central data store that can be only changed by messages from the view, whereas Reflex and Concur are more flexible, allowing both centralized and component-local state. A common complaint regarding Reflex is that there is no recommended application architecture - it errs on the other side of the flexibility vs. best practices spectrum.

As for **routing**, Miso has routing built into its base library. There are several gattempts at a routing library in Reflex, though the situation is the same as with templating libraries. Concur with its small ecosystem does not have routing at all, it would be necessary to implement form scratch for a production-ready application.

In **forms** - and UI components in general - the selection is not good. There are several components collections for Reflex which use popular CSS frameworks (Bootstrap, Semantic UI), though each has many missing pieces and they lack components that need to be re-implemented anew in each application - forms in particular. Miso and Concur do not have any publicly available UI component libraries, or at least none that I was able to find.

**Accessibility** as a whole has not been a focus of Web development in Haskell. It is possible to reuse JavaScript accessibility testing tools however, though I have not seen any sort of automated testing done on any of the publicly available Haskell applications. The only area with continued developer focus is **loading speed**, as the size of build artifacts was a problem for a long time. That has been ameliorated to the level of a common JavaScript application however, so that is not a critical concern. **Prerendering** is also supported by Miso and Reflex, which helps speed up load times as well.

Moving on to the topic of **build tools**: there are three main options in Haskell - Cabal v2 [4], Stack [6], and Nix. There are also other options - Snack [24], aiming for the best of these three but not yet ready for production use, or Mafia [38], which is not too popular in the community at large. Cabal is the original Haskell build tool which gained a bad reputation for some of its design decisions (the so-called 'Cabal hell'), though most of them were fixed in 'Cabal v2' which puts it on par with its main competitor, Stack. Stack tried to bring Haskell closer to other mainstream programming language by introducing several new features like automatic download of the selected compiler or a curated subset of the main Haskell package repository, Stackage. It succeeded in that, becoming the tool of choice for a large part of the Haskell community in the process. Nix, as mentioned in the previous section, is a general-purpose build tool and not a Haskell-specific one. It has very good cross-compilation capabilities, however, which is the reason it is especially used for frontend Haskell.

Glasgow Haskell Compiler (GHC) is the main Haskell **compiler** used for the creation of native binaries. Compilation to JavaScript, as required for frontend development, is supported by a separate compiler, GHCJS, which uses GHC as a library. Setting up a GHCJS development environment with Cabal is not a trivial process and using Stack limits the developer to old GHC versions, so it is Nix that is usually recommended. When set up correctly, Nix offers almost a one-click setup, downloading the compiler and all dependencies from a binary cache or compiling them if unavailable. Reflex especially, in the reflex-platform [33] project, uses the cross-compilation capabilities of Nix to allow applications to compile for Android, iOS, desktop, or the web simultaneously.

The main problem of GHCJS has been speed and the size of the produced JavaScript. The latter has been gradually improving and is now mostly on par with modern JavaScript framework, the former is harder to improve though, and GHCJS applications are still within a factor of 3 of native JavaScript ones [29]. However, this should be improved soon by compiling to WebAssembly instead of JavaScript. There are two projects trying to create a Haskell-to-WebAssembly compiler in parallel - Asterius [42], and WebGHC [12]. They are so far in alpha, but I expect them to be production-ready by the end of 2019.

Moving on to the topic of **debugging tools**, this is where Haskell on the frontend is lacking the most. While it is possible to use the browser's built-in DevTools and their debugger and profiler, the compiled output of GHCJS does not correspond to the original Haskell code too much, which makes using the debugger quite hard. There are no other debugging tools, though in my experience I did not ever feel the need to use anything else than writing debugging output to the console.

In contrast, there are many **quality assurance** tools available for Haskell in general, of which almost all are available for use in frontend development. Starting with static quality assurance, Hlint is the standard 'linter' for Haskell, well-supported and mature. There are several code formatters, Hindent is the most widely used one, which enforces a single style of code as is common in other contemporary languages (e.g. gofmt for Go). As for test frameworks, there are many options. HSpec or HUnit are examples of unit- or integration-testing frameworks, property-based testing is also common in Haskell, with QuickCheck [5] being the most well-known example. For end-to-end testing in the browser, there are libraries that integrate with Selenium.

Haskell has a quite bad reputation for the lack of **editor integration**. The situation is better with the recent Language Server Protocol project, where haskell-ide-engine, Haskell's language server, enables users to write Haskell in contemporary editors like Atom easily. The language server supports type-checking, linting and formatting, and also common IDE features like 'go-to-definition' or 'type-at-point'.

Compiling applications as **mobile or desktop apps** is well-supported in Reflex, though not in Miso or Concur. Using the scaffolding of reflex-platform makes supporting different platforms almost automatic, as Nix takes care of switching between compilers: GHCJS for the Web, regular GHC for the desktop and cross-compiling GHC for iOS or Android. Bundling the compiled applications for distribution for each platform is a bit more involved, though there are efforts to automate even that.

**Code generators** are quite limited in Haskell. Stack has a templating system for new project initialization, though there are no templates for frontend development so far. Cabal comes with a single standard template for a blank project but lacks customization options for creating framework-specific templates. And Nix does not do code generation at all. The common practice so far is to make copy of a repository containing the basics, edit project-specific details, and use that as a base for a new project. I have not found any attempts at component generation in Haskell.

The last point I want to mention is **documentation**. It is generally agreed that it is Haskell's weakest point - despite having a standardized high-quality tool for creating API documentation (haddock), writing it is often an afterthought, with even commonly used packages having no documentation at all or written in such a way that a new user has no choice but to study its code to understand the package. In this work, I will strive to avoid this common flaw.

## 3.4 Implementation plan

In this section, I will use the terminology used in the paper "Evolving Frameworks" [34] to describe the work performed in the rest of this work and follow-up work as well. The paper describes common stages that frameworks take as they develop. While is uses terminology from object-oriented frameworks, most of the concepts apply just as well In Haskell.

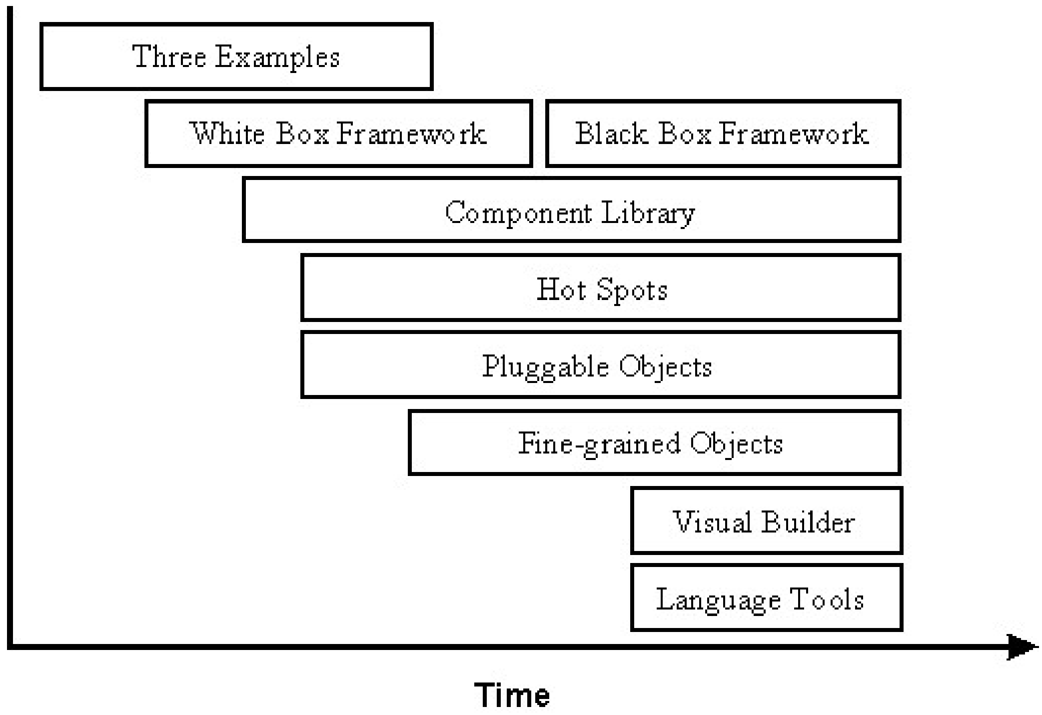


Figure 3.1: The timeline of patterns as described in Evolving Patterns

To briefly describe the terms and how they relate to this work:

- **"Three Examples"** are three applications from which the framework will draw common themes and architecture, so that it fulfills real-world needs. This is what we will go through in the next section, where we take three existing application specifications and build a Haskell version of it.
- In a **"White Box Framework"**, the architecture is extracted into a separate library and expanded or re-implemented in further applications. The author emphasizes 'programming-by-difference', where the programmer extends library code and later factors out commonly repeated patterns into the library. In this work, this is the approach taken after implementing the "Three Examples" to create the basics of the shared libraries.
- The next patterns, "**Component Library**", "**Hot Spots**", and "**Pluggable/Fine-grained Objects**" are all an extension of the above, focusing on extracting concrete

components and restructuring the architecture to improve developer experience in specific ways. This level, nor the further ones are not implemented in this work.

- Skipping a "**Visual Builder**", which is not a common pattern in Web frameworks, there are some basic "**Language Tools**" implemented as a part of creating the libraries, namely a debugging console for watching specific values and an inspector of the application storage. [TODO: specify after implementing]

Not mentioned as a part of the patterns but also an essential part of framework development is thorough documentation and guides, as well as test coverage of library code, which is also done as a part of the work on libraries in the latter parts of this work.

The above is a quite general description, so we will now enumerate the specifics of the implementation plan, starting with a reiteration of the requirements of a PWA from the introduction, which is the end goal of this work.

- Pages are responsive on tablets & mobile devices
- All app URLs load while offline
- Metadata provided for Add to Home screen
- Page transitions do not feel like they block on the network
- Each page has a URL
- Pages use the History API
- Site uses cache-first networking
- Site appropriately informs the user when they are offline
- Push notifications (consists of several related requirements)

There are, however, several components missing in the Haskell ecosystem that need to be created from scratch:

- A full-featured browser routing library. While there are some existing implementations, they are either incomplete or long abandoned.
- A wrapper around ServiceWorkers, or a template to simplify project creation.
- A push notifications library. This will need to be both a server-side library, for creating them, and a client-side consumer, to parse them.
- A way to prerender the application - either just the HTML "app shell" or all pages on the site.
- An offline storage library for the client. Here are several possible variants, in the order of difficulty:
  - plain storage datatype with LocalStorage, SessionStorage, and IndexedDB backends

- a storage including a transparent cache integrated with the network layer
- a storage with an invalidation or auto-refresh functionality, using an event stream from the server
- a storage with offline-capable synchronization capabilities

These components do not comprise a fully integrated framework in the sense of e.g. Angular, such frameworks are quite uncommon in the Haskell ecosystem. More common are collections of libraries that play well together, where one library provides the fundamental datatype - the "architecture" of the application - and other libraries fill in the functionality, which is what we will work on. Of the proposed components, only the routing library is an "architectural" one in the sense that it will influence the shape of the application and its fundamental data types.

# Chapter 4

# Components

TODO: Demonstrate the principles of components on 'src-snippets' code, where I will show the smallest possible code that implements that functionality

## 4.1 Component A

### 4.1.1 Design

### 4.1.2 Implementation

### 4.1.3 Testing

### 4.1.4 Other options, possible improvements

# Chapter 5

# Applications

## 5.1 Workflow and tools

TODO: describe the development flow of an app built using these tools

- starting out - three layer cake & esp. the inner one
- QA (tests, e2e, CI, . . . ), documentation
- development tool options
- deployment options

## 5.2 TodoMVC

There is an abundance of web frameworks, and there are several projects that aim to give developers a side-by-side comparison of them. Out of these, the original and most well-known one is TodoMVC [37], which is aimed at "MV* frontend frameworks". There are currently 64 implementations of their specification - some of them are variants of the same framework though. There are a few others - HNPWA is aimed at Progressive Web Applications and it is a tad smaller, with 42 implementations. The last comparison project selected for this work is RealWorld. This one has both a frontend and a backend part and there is also a small number of full-stack frameworks. It offers a quite thorough comparison, with 18 frontends, 34 backends, and 3 full-stack implementations.

We will start with TodoMVC as it is the simplest of the three. TodoMVC is, as the name hints, a web application for managing a to-do list. It is not a complex project but it is intended to exercise fundamental features of a framework - DOM manipulation, forms and validation, state management (in-memory and in LocalStorage), and routing.

## 5.3 HNPWA

HNPWA [36] is a client for Hacker News, a technological news site. Unlike TodoMVC, HNPWA does not provide a rigid specification and consists only of a rough guideline of what to implement. The task is to create a Progressive Web Application that displays information from a given API. The application must be well optimized (to achieve score 90 in the Lighthouse tool) with optional server-side rendering.

## 5.4 RealWorld

RealWorld [40] is the most complex of the comparison projects. It is a clone of Medium, an online publishing platform, so it requires everything a "real world" application would. The task is split into a backend, defined by an API specification, and a frontend, defined by an HTML structure.

There is a number of features the application needs to support, namely: JWT (JSON Web Token) authentication with registration and user management, the ability to post articles and comments, and to follow users and favorite articles.

# Chapter 6

# Conclusion

TODO: return to the comparison with JS, PHP, ... frameworks

TODO: describe possible follow-up work, what I will be working on - define specific topics and make concrete examples

– The final chapter includes an evaluation of the achieved results with a special emphasis on the student's own contribution. A compulsory assessment of the project's development will also be required, the student will present ideas based on the experience with the project and will also show the connections to the just completed projects. [16]

# Bibliography


[1] Ashkenas, J.: CoffeeScript hits 1.0 – Happy Holidays, HN | Hacker News. Retrieved from: https://news.ycombinator.com/item?id=2037801

[2] Bar, A.: What Web Can Do Today. Retrieved from: https://whatwebcando.today/

[3] Bernardy, J.-P.; Boespflug, M.; Newton, R. R.; et al.: Linear Haskell: practical linearity in a higher-order polymorphic language. *arXiv preprint arXiv:1710.09756*. 2017.

[4] Cabal Team: The Haskell Cabal. Retrieved from: https://www.haskell.org/cabal/

[5] Claessen, K.; Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*. vol. 46, no. 4. 2011: pp. 53–64.

[6] Commercial Haskell: The Haskell Tool Stack. Retrieved from: https://docs.haskellstack.org/

[7] Czaplicki, E.: Elm: Concurrent FRP for Functional GUIs. 2012.

[8] Dolstra, E.: *The purely functional software deployment model*. Utrecht University. 2006.

[9] Dolstra, E.; Löh, A.: NixOS: A purely functional Linux distribution. In *ACM Sigplan Notices*, vol. 43. ACM. 2008. pp. 367–378.

[10] Eisenberg, R. A.: Dependent types in haskell: Theory and practice. *arXiv preprint arXiv:1610.07978*. 2016.

[11] Facebook: ReactJS. Retrieved from: https://reactjs.org/

[12] Fancher, W.: WebGHC. Retrieved from: https://webghc.github.io/blog

[13] Flipkart: Flipkart. Retrieved from: https://www.flipkart.com/

[14] Google: Angular. Retrieved from: https://angular.io/

[15] Google: Progressive Web App Checklist | Web.
Retrieved from:
https://developers.google.com/web/progressive-web-apps/checklist

[16] Hlavsa, Z.; et al.: *Pravidla českého pravopisu*. Academia. 2005. ISBN 80-200-1327-X.

[17] Hudak, P.; Hughes, J.; Peyton Jones, S.; et al.: A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-766-7. pp. 12–1–12–55. doi:10.1145/1238844.1238856.
Retrieved from: http://doi.acm.org/10.1145/1238844.1238856

[18] Jain, A.: ajnsit/concur: An unusual Web UI Framework for Haskell.
Retrieved from: https://github.com/ajnsit/concur/

[19] Johnson, D.: dmjio/miso: A tasty Haskell front-end framework.
Retrieved from: https://github.com/dmjio/miso/

[20] Jones, S. P.: *Haskell 98 language and libraries: the revised report*. Cambridge University Press. 2003.

[21] Lab, Q. F.: Queensland FP Lab - Home.
Retrieved from: https://qfpl.io/

[22] Loder, W.: *Web Applications with Elm: Functional Programming for the Web*. Apress. 2018. ISBN 9781484226100.
Retrieved from: https://books.google.cz/books?id=KnhqDwAAQBAJ

[23] Marlow, S.: Fighting spam with Haskell. 2015.

[24] Mattia, N.: nmattia/snack: Nix-based incremental build tool for Haskell projects.
Retrieved from: https://github.com/nmattia/snack/

[25] McKenzie, S.: 5.0.0 Released · Babel.
Retrieved from: https://babeljs.io/blog/2015/03/31/5.0.0

[26] Microsoft: Language Server Protocol Specification.
Retrieved from:
https://microsoft.github.io/language-server-protocol/specification

[27] Microsoft: TypeScript - JavaScript that scales.
Retrieved from: https://www.typescriptlang.org/

[28] Mozilla: Using Service Workers - Web APIs | MDN.
Retrieved from: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers

[29] Nanda, S.: Benchmarks: GHCJS (Reflex, Miso) and Purescript (Pux, Thermite, Halogen).
Retrieved from: https://medium.com/@saurabhnanda/benchmarks-fp-languages-libraries-for-front-end-development-a11af0542f7e

[30] Nanz, S.; Furia, C. A.: A Comparative Study of Programming Languages in Rosetta Code. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. May 2015. doi:10.1109/icse.2015.90.
Retrieved from: http://dx.doi.org/10.1109/ICSE.2015.90

[31] Obsidian Systems: reflex-frp/reflex.
Retrieved from: https://github.com/reflex-frp/reflex/

[32] Obsidian Systems: reflex-frp/reflex-dom.
Retrieved from: https://github.com/reflex-frp/reflex-dom/

[33] Obsidian Systems: reflex-frp/reflex-platform.
Retrieved from: https://github.com/reflex-frp/reflex-platform/

[34] Roberts, D.; Johnson, R.; et al.: Evolving frameworks: A pattern language for developing object-oriented frameworks. *Pattern languages of program design*. vol. 3. 1996: pp. 471–486.

[35] Systems, O.: Obsidian Systems | Cutting edge software solutions.
Retrieved from: https://obsidian.systems/

[36] TasteJS: HNPWA.
Retrieved from: https://hnpwa.com/

[37] TasteJS: TodoMVC.
Retrieved from: http://todomvc.com/

[38] The Haskell Mafia: haskell-mafia/mafia: Provides protection against cabal swindling, robbing, injuring or sabotaging people with chopsticks.
Retrieved from: https://github.com/haskell-mafia/mafia/

[39] The JAMstack team: JAMstack - JavaScript, APIs, and Markup.
Retrieved from: https://jamstack.org/

[40] Thinkster: gothinkster/realworld: "The mother of all demo apps".
Retrieved from: https://github.com/gothinkster/realworld

[41] Tweag: Tweag I/O - software innovation lab.
Retrieved from: https://tweag.io/

[42] Tweag: tweag/asterius: A Haskell to WebAssembly compiler.
Retrieved from: https://github.com/tweag/asterius

[43] Twitter: Twitter.
Retrieved from: https://mobile.twitter.com/home

[44] Uber: Uber.
Retrieved from: https://m.uber.com/looking

[45] You, E.: Vue.js.
Retrieved from: https://vuejs.org/

# Appendices

# Appendix A

# Contents of the attached data storage

TODO: fill in

# Appendix B

# Poster

TODO: fill in