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 2, 2019

Contents

Chapter 1

Introduction

Let's imagine we want to create a new e-shop, with all the conveniences that customers are nowadays used to - [TODO: . . .], a notification when the order is ready, and ideally an offline-available overview of the order for when the user is picking it up. What would be the easiest way to implement such a service? If we want to show something offline, we need a mobile application, the same for notifications. But [TODO: research about mobile engagement] and downloading a mobile application means another dozen megabytes.

That's what the new trend of Progressive Web Applications (PWAs) is trying to solve. Fast - no need to wait for page reloads - and reliable - service workers help overcome problems with unreliable internet, both for page load (by caching the app itself, and in-app caching of app-specific data) and for background sync of user-modified data.

My language of choice is Haskell though, for both frontend and backend development, and while writing Haskell applications for the browser is possible, creating a Progressive Web Application requires more than just the ability to manipulate the DOM. That is what this work is about - filling in the gaps and creating all the prerequisites for writing Progressive Web Applications in Haskell. We will start by identifying what is exactly is lacking in the Haskell ecosystem and design and implement those missing pieces. Afterwards, we'll walk through the process of creating a PWA from prototype to production and see what else could be made easier and more straightforward as a follow-up to this work.

1.1 Related work

While using Haskell in the browser is not a common choice, it is still a quickly growing niche. It is a relatively recent one as GHCJS, the Haskell-to-JavaScript compiler, has been in development since 2010 [TODO: gh repo link] but it had a production-ready version only in 2013 [luite's blog]. It is not the target of academic work, but there are several commercially sponsored projects. To name a few, Reflex and Obelisk from Obsidian Systems [TODO: link], Asterius from Tweag, and many learning materials from QFPL [TODO: their github].

While it's a quickly growing area, it's still not an established one, and there are many libraries and tools still missing, ones that developers used to other languages have learned to expect and require.

Chapter 2

Technologies

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

2.1 Progressive Web Application

"Progressive Web Application" is a term that covers several relatively new technologies. It is the continuation of the general trend of expanding the capabilities of browser applications and closing the gap between them and native mobile applications. While many of these technologies apply also on desktop, the main target audience is mobile - [TODO: some statistics].

The technologies are:

- Web App Manifest - a specification for the centralized location of application metadata - its name, icons, display mode, ...
- Service Workers - "a scriptable network proxy" [TODO: as per Wiki]. 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 [TODO: `https://whatwebcando.today/`] - examples include geolocation or audio/video capture

What is a Progressive Web Applications exactly is defined by a checklist created by Google, the `https://developers.google.com/web/progressive-web-apps/checklist` [TODO: ref]. It describes two levels of PWAs, a 'Baseline PWA' and an 'Exemplary PWA'. The main defining characteristics of a Baseline PWA are: served over HTTPS, responsive design, all URLs available even while offline, and non-blocking page transitions. While there are more requirements for an Exemplary PWA, these are the most important ones.

Some more notable PWAs are: Instagram, Twitter [TODO: refs, see wiki], Uber, Flipkart, or Tinder.

2.2 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"
```

The programming language Haskell is a rather old one, despite spreading into the industry only recently, having been around since 1990 [TODO: reference]. It is a language that started out as a basis for research into the design of functional languages. It has served as such, and in fact it still is the target of active research - some more prominent projects are "Dependent Haskell" [TODO: ref] and "Linear Haskell" [TODO: ref].

Haskell is described as a "statically typed, purely functional programming language with type inference and lazy evaluation" [TODO: ref]. It is a language in which its expressive type system enables very precise control over TODO: expressive types, dialog with the compiler, types encoding effects, citation for "If it compiles, it works".

Why Haskell? It is a strongly typed language, that helps ensure correctness and prevent undefined behaviors - but it is less verbose and more expressive than Java and other languages a developer would imagine as 'strongly typed'. Haskell is one of the many languages that can be used in the browser - not directly, but by compiling down to JavaScript. Another technology that enables languages to run in the browser is WebAssembly, an alternative assembly language and a runtime designed specifically for the Web. Compiling Haskell for the Web via WebAssembly is almost doable, there are two active projects creating a Haskell compiler backend - WebGHC [TODO: ref] and Asterius [TODO: ref].

Compile-to-JavaScript languages aren't as rare as it may seem. While languages that aren't based on JavaScript itself aren't exactly common, web developers have been using JavaScript compilers for a long time - CoffeeScript is rather popular language announced in 2010 [TODO: ref], and developers wanting to use new ECMAScript 6 or 7 features (now supported in most browsers) also had no choice but to use compilers [TODO: ref Babel history].

It is a language that enables its users to write reliable software - it eliminates entire classes of programming errors [TODO: ref, enumerate]. The errors that remain even after the program successfully compiles are usually logic or conceptual errors.

While Haskell is not a language commonly associated with frontend development, it is one of the many languages with the ability to use JavaScript as the compilation target, instead of plain assembly or LLVM. In fact, such languages have now become quite common in frontend development [TODO: ref], as is exemplified by the rapid rise of TypeScript, a superset of ECMAScript 6 [TODO: ref], or Elm, a framework with its own language based on Haskell [TODO: ref].

Of the many reasons for selecting a language other than JavaScript for frontend development, one of the more notable ones is the ability to share code between the server and its client in the case they are written in the same language. This is the basic idea of the framework Meteor [TODO: ref], and in fact the ability to run 'isomorphic code' - the same code on the client and the server both - is its main marketing point.

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 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
  };
  inherit perl;
}
```

One technology that wasn't yet mentioned but that will support our entire build process - from compiling to deploying - is Nix. Nix [TODO: ref homepage] is a package manager with focus on reproducibility and isolation. It is described as a purely functional package manager, where every package is built by a function without side-effects, with the result being immutable. Nix also ensures that the exact version of dependencies is used even during runtime - all the way up to `libc`.

Nix is a declarative build tool, 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 [TODO: refs]. Nix shines at cross-compilation, which is the main I'll use it in this thesis - compiling to JavaScript or Android/iOS is trivial after the initial setup.

Nix is another rather old technology actively developed since 2004 after Eelco Dolstra developed this idea in his academic work [TODO: refs]. One package consists of a closure of all of its runtime dependencies, so even packages using different versions of dynamically linked libraries or even libc can coexist on the same machine. Adding atomic deployments and rollbacks is then quite easy, as a user environment only consists of symbolic links to the read-only Nix store - that is very useful for NixOS or NixOps.

Chapter 3

Research

As this work doesn't 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 very small API, whereas Angular provides a quite opinionated platform, rather than a framework. Individual frameworks are very 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 several features that a framework needs to take care of - **templating** is one of them. 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 very 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.

Supporting **internationalization** in a framework is easier - not to implement, but simple to package as a library. 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, but doesn't need to fetch anymore data. There is another variant of "server-side rendering" called the "JAM stack" pattern, 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 doesn't 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 tool, 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 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 Wis also important in some ecosystems, which includes common IDE features like auto-completion or refactoring tools. The situation here is quite good lately, with the new Language Server Protocol (LSP) playing a big role in enabling editors to support a wide variety of languages. There are some parts of editor support that can be framework-specific - supporting an embedded DSL, integrating framework-specific debugging tools, ...

While we were talking about Web frameworks so far, some of them support not only the browser runtime environment but can be packaged as a **mobile app** for Android or iOS, or a **native desktop application** the many desktop operating systems. For mobile support, frameworks either provide wrappers around Apache Cordova, but some go further and support native mobile interfaces (React Native). For desktop support it isn't Cordova but Electron which is the basis here. The main benefit of packaging an application specifically for a platform is performance, as they are usually faster to load and to use. There are other benefits, like access to device-specific APIs or branding.

The last point in this section is **code generators**. of which there are two variants: project skeleton generators, which are provided in a large majority of frameworks and which include all files for a project to compile and run. 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 and React. Vue.js 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 have 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', with 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 by having a recommended set of libraries for common use cases, but that doesn't help with another complaint called the 'JavaScript fatigue', where libraries come and go each year, where the common belief is that you're missing out on opportunities if you're not learning at least one new framework per year.

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, but this 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.

For many of the topics mentioned in the previous section - routing, forms, build tools, mobile and desktop applications - it is true that they are built into Angular, and that there are many available libraries for React and Vue. In my investigation, I haven't 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' [TODO: ref] as their basis, as traditional event-based imperative programming does not really fit those criteria.

There are five production-ready UI toolkits for the Web that I have found. Out 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 [TODO: ref] (and Reflex-DOM [TODO: ref], its DOM bindings) seem like the most actively maintained and developed ones. Reflex is also sponsored by Obsidian Systems [TODO: ref] and the most popular one in the Haskell community, so its future seems promising. Reflex follows the traditional FRP with events and behaviors, adding 'dynamics' to the mix, and building a rich combinator library on top of them.

Miso [TODO: ref] was built as a re-implementation of the 'Elm architecture' in Haskell. It uses a very strict form of uni-directional dataflow with a central data store at the one side, and the view as a function from the state to a view on the other, passing well-defined events from the view to the store.

Concur [TODO: ref] tries to explore a different paradigm, which tries to combine 'the best of parts' of the previous two approaches. So far it has a very small range of features, focusing on exploring on how this paradigm fits into browser, desktop or terminal applications. It is a technology I intend to explore in the future when it's more mature, but that doesn't seem viable for a large-scale application, at least compared to its competitors.

TODO: examples of Reflex, Miso, Concur

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, but they are incomplete or unmaintained. 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).

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 generic, allowing for both centralized and component-local state. A common complaint in Reflex is that there is no recommended architecture to the application - it errs on the other side of the flexibility vs. best practices spectrum.

As for **routing**, Miso has routing built into its base library. In Reflex, there are several attempts at a routing library, but not a universally accepted solution. 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 very good. For Reflex, there are a few collections of components that use popular CSS frameworks (Bootstrap, Semantic UI), but there are many missing pieces that are re-implemented in each application - forms in particular. Miso and Concur do not have any publicly available UI component libraries (or at least none that I have been 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, but I haven't 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 [TODO: ref], Stack [TODO: ref], and Nix [TODO: ref]. There are also other options - Snack [TODO: ref], aiming for the best of these three but not yet ready for production use, or Mafia [TODO: ref], 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'), but 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 with several new features like automatic download of the selected compiler or a curated subset of the main Haskell package repository, Stackage. Nix is a not a Haskell-specific build tool but rather a general-purpose one, but it has very good Haskell support and cross-compilation capabilities, which is the reason it is especially used for frontend Haskell.

Glasgow Haskell Compiler (GHC) is the main Haskell **compiler** used for native compilation. 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 cross-platform one-click setup, downloading the compiler and all dependencies from a binary cache or compiling them if unavailable. Reflex especially, in the reflex-platform [TODO: ref] 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 compiled JavaScript. The latter has been gradually improving and is now mostly on par with modern JavaScript framework. The former though is harder to improve and GHCJS applications are still within a factor of 3 of native JavaScript ones [TODO: ref bench]. However, this should be improved soon by compiling to WebAssembly instead of JavaScript. There are two projects working on this in parallel - Asterius [TODO: ref], and WebGHC [TODO: ref]. So far they are 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 very much, which makes using the debugger quite hard. There are no further debugging tools. In my experience though, I didn't ever feel the need to use more than tracing 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 very common in Haskell - with QuickCheck 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 (LSP) project, where haskell-ide-engine (HIE), Haskell's language server, enables users to write Haskell in contemporary editors like Atom easily. HIE 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 (but not in Miso or Concur). When using the Nix scaffolding of reflex-platform, compilation automatically switches between GHCJS (for the Web), regular GHC (for the desktop) and cross-compiling GHC (for iOS or Android). Bundling the compiled applications for distribution a bit more involved, but most of the process is automated.

Code generators are quite limited in Haskell. Stack has a templating system for new project initialization, but there are no templates for frontend development. Cabal comes with a single standard template for a blank project, but it does not have an option for creating frontend-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.

Last point I want to mention is **documentation**. It is generally agreed that documentation is Haskell's weakest point - despite having a standardized high-quality APIdoc tool (haddock), 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

I will use the nomenclature from the "Evolving Frameworks" paper [TODO: ref] to describe my my goals. [TODO: describe the general outline of that paper] Long-term, I aim to go from "Three Examples", a place to look for commonly repeated patterns, through a "White-Box Framework", a general structure/architecture of an application wrapped into a library, through a "Component Library", when that library will be extended with commonly duplicated functionality, all the way to "Pluggable Objects", where the framework provides most of the commonly used functionality so that application logic is the only thing missing from a finished prototype.

TODO: include the Evolving Frameworks image

Building an integrated platform is not my primary goal - it is hard for a new and opinionated platform to succeed in the Haskell ecosystem. In this thesis, I expect to do the first step only - create a few applications, fill in all the missing parts in the Haskell ecosystem that aren't covered by existing libraries, and extract a common application skeleton, a set of libraries, and a set of guides or tutorials that make it is easy go from the skeleton to a working prototype of a PWA.

The goal of this work is to make it possible to create Progressive Web Applications. To reiterate the description from the introduction, these are the requirements:

- Pages are responsive on tablets & mobile devices
- All app URLs load while offline
- Metadata provided for Add to Home screen
- Page transitions don't 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're offline
- Push notifications (consists of several related requirements)

To get there, there are several features that aren't covered by any existing Haskell library or tool. This is a list of the required parts:

- 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 'shell' or all pages on the site.
- An offline storage library for the client.

There are several variations of the last point with different levels of difficulty. Only the first variation is required, with the other ones being a part of follow-up work of this thesis.

- 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 don't comprise a fully integrated 'platform' in the sense of e.g. Angular - those 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. 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'll 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, 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 that I've selected 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 should exercise the basics of a framework - DOM manipulation, forms and validation, state management (in-memory and in LocalStorage), and routing.

http://todomvc.com/

5.3 HNPWA

HNPWA 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.

https://hnpwa.com/

5.4 RealWorld

RealWorld 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.

https://github.com/gothinkster/realworld

Chapter 6

Conclusion

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

TODO: describe possible follow-up work, what I'll 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. [1]

Bibliography

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

Appendices

Appendix A

Contents of the attached data storage

TODO: fill in

Appendix B

Poster

TODO: fill in