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
February 1, 2019

{

Contents

}

Chapter 1

Introduction

- a concrete motivating example of a Haskell app that needs a web client, plus why to write a PWA
- general trends in Haskell and frontend development
- related work (Tweag, Obsidian, ...)

Chapter 2

Technologies

We'll first have a more detailed look at the above-mentioned technologies, [...]

2.1 Haskell

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".

TODO: language to write reliable software in - it eliminates entire classes of programming errors - usually the ones that remain are logic 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.

TODO: Haskell on the frontend is a rapidly growing area with many interested companies. The main projects in this area are sponsored and developed by them - some of the main contributors are: Obsidian (Reflex, Obelisk), Tweag (Asterius, Inline-js), and QFPL (several UI components). [TODO: refs] 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.

```
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"
```

2.2 Nix

- *"Nix package manager, a "purely functional" package and configuration manager for computer systems"* – Wikipedia
- *"Describe your end result, then magic happens"*
- Tools build on top:
  - Nix = package manager
  - NixOS = operating system
  - NixOps = cloud deployment tool
- cross-compiling
- since 2004, Eelco Dolstra's Ph.D. thesis in 2006
- purely functional, lazy evaluation
- one program consists of a closure that includes all dependencies including libc
- atomic upgrades, rollbacks
- complete isolation of dependencies, no DLL hell
- NixOS = OS built on top of Nix
- NixOps = a cloud deployment tool
  - conceptually: Terraform/CloudFormation + Puppet/Ansible
  - a network specification -> magic -> running set of servers on AWS, VPSs, VirtualBox, ...

```
{
  network.description = "Web server";

  webserver = { config, pkgs, ... }: {
    services.httpd.enable = true;
    services.httpd.adminAddr = "alice@example.org";
    services.httpd.documentRoot =
      "${pkgs.valgrind.doc}/share/doc/valgrind/html";
    networking.firewall.allowedTCPPorts = [ 80 ];

    deployment.targetEnv = "virtualbox";
  };
}
```

2.3 PWA

- concrete examples of existing apps
- general reasons for PWA development
- Web Platform APIs, Service Workers

Chapter 3

Research

In order to write a SPA/PWA, there are some tools and supporting libraries that a developer can't live without. We'll first walk through a high-level overview of the common features of the Web frameworks of today, then go through the most popular frameworks and see where they shine. Afterwards, we'll have a look at the ecosystem of Haskell and try to find its strong and weak points, and especially the places where the tools or libraries are entirely missing.

3.1 Common features of Web frameworks

– A description of the things I'll be looking for, partially from Wikipedia's Comparison of web frameworks and Comparison of JavaScript frameworks, partially from my own experience, and partially from the feature lists of the frameworks that I'll be looking into.

3.1.1 Tooling

I'll start with the things you encounter first when setting up a project, its tools. Developers have wildly differing levels of expectations from their tools. A Python developer might expect just a text editor and an interpreter, whereas a JVM developer might not be satisfied with anything less than a full-featured IDE.

Code generators or scaffolding tools start with creating a package manifest and a `src/` directory, going on to generators that set up a few different types of projects based on templates, all the way to tools that can add an entire website module, perhaps even with database migrations.

Build tools range from a set of conventions on how to use your build tool that might get formalized in your Makefile, through a CLI tool that takes care of building, testing and perhaps even deploying your project, to the way of the IDE where anything you can think of is just a few clicks away.

Debugging tools also come in many flavors and for many purposes. On the side of the server, you have all the usual tools for the language, plus a few more - a toolbar with an overview of everything that goes on in a page render or an AJAX call, or the option to

remotely connect to a running process and to debug live. Client-side, we have the now irreplaceable DevTools with a built-in debugger and profiler, but some frameworks go even further and provide a framework-specific tools - React's component tree, or Elm's time-traveling debugger.

Quality assurance tools have many sub-categories. From static code analysis tools or linters, as they are commonly known; through tests - unit, integration, end-to-end tests, or more exotic ones like marble tests or visual regression tests; to profilers - runtime or allocation measurements, frontend performance measurements, or more involved tools like performance evolution tracking.

3.1.2 Features

TODO: Reread and clean up

Templating, which at the frontend means a way to compose the HTML that makes up an application, usually including some render logic and variable interpolation. In some frameworks the whole program is a template (see React), some have templates in separate files and pre-compile them during runtime (see Angular). Templates sometimes contain CSS as well (see the new CSS-in-JS trend)

Forms are the basic element of interactive applications. 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, manual or automatic. Under 'forms' I count a way to render a form, to validate user input, and collect the result.

Routing means manipulating the displayed URL using the History API and changing the application state to reflect it, and also loading the correct state on application start-up. This can also include animated transitions between pages.

Internationalization has many levels. At its most basic, it means simple string translations, pluralization, and word order. Going further, it means also RTL scripts, date/time formats, currency, or time zones.

Modern web frameworks also provide a way to use a web application's code and compile it into a **native mobile application**. It's commonly implemented as a wrapper application around web apps using Cordova, but technologies like React Native go further and use native UI elements. Mobile applications can be faster to use and faster to load, can access device-specific APIs not exposed via Web Platform APIs.

Some frameworks also support creating a **desktop application** that reuses a web application's code. The underlying technology here is Electron and not Cordova, but the benefits are the same - greater speed and access to device-specific functionality.

3.1.3 Accessibility (ARIA, ...)

- the key word now is ARIA = support for screen readers
- also, semantic elements, text contrast, customization

- also also, keyboard-accessibility (shortcuts, every clickable element accessible via keyboard = tabindex)
- accessibility testing (automatic as well, see aXe)

3.1.4 Optimistic updates

- one of the things I want to focus on
- broadly, expecting that every network request will be successful and updating the GUI accordingly
- rolling back app state in case of failure, with notifications

3.1.5 Web Platform

- location, camera, touch, vibration, ...

3.1.6 Pre-render

- one approach to shortening start-up times
- serving HTML with all the content already inside, no need for more requests to the backend for the initial page load
- JS takes over and uses what's already been loaded
- can be static or dynamic:
  - static = JAM stack, serving a bunch of files rendered at compile-time
  - dynamic = rendering the HTML at runtime

TODO: What about the backend feature write-ups I have?

3.2 JavaScript ecosystem

3.2.1 Angular

TODO: what is it? On a first look, Angular looks like a well thought-out frontend framework. Written in Typescript with comprehensive documentation and great tooling, it seems that the authors have learned from their mistakes with AngularJS.

Some notable features:

- command line tool, `ng` - it streamlines setting up the entire project - scaffolding, preparing build and testing tools, starting a development server, ...

- runtime environments - from server-side rendering, PWAs with ServiceWorkers, to native and desktop applications, it seems that Angular tries to cover every possible use-case
- tooling other than the `ng` tool - browser extensions for runtime debugging, IDEs and others. I haven't thought of a tool I would miss, but I'm used to minimalism in tooling from the Haskell world...

Some negatives that developers complain about:

- Angular is intimidating for a new developer, it's too complex and there's a lot to learn
- Too much 'magic' - related to the previous point, there's a lot of abstraction and it's not easy to understand all the layers
- Code bloat - the amount of boilerplate and also the size of the resulting bundle
- Too opinionated - if you don't like 'the angular way', you're out of luck here
- scattered documentation - too many articles and tutorials out there for AngularJS that can't work with the new Angular

3.2.2 React

React is not a framework in itself. Rather, it's a library that focuses on a single thing and does it in a unique enough way that there's sprung up an entire ecosystem around it. In it, there are groups of libraries that build upon React, each focusing on a single feature - UI components, state management, forms etc.

There's a large jungle of libraries, each one with a different scope and focus. Choosing a library that fits your problem can sometimes take many attempts. Add to it the fact that libraries, frameworks and tools come and go quite quickly - the main cause of the so-called "JavaScript fatigue - and the fact that in JavaScript, it's fashionable to write extremely small libraries, and you have a recipe for a quite unpleasant development experience.

I'll try to go through some of the most popular 'frameworks' that build on React, though each one is more of a pre-built toolkit of libraries and tools and bits of glue in between, rather than cohesive frameworks. In general, the React world is a lot more mix-and-match than developers used to enterprise frameworks would expect.

Create-react-app, nwb, Razzle, and Neutrino all cover only the build process. Next.js is the first one that I've found that goes a step beyond just pre-configuring Webpack and other build tools - it provides other features that are starting become standard - server-side runtime rendering, link prefetching, and build-time prerendering. It's also the first tool I found that considers that a website can consist of multiple applications, via its 'zones' feature.

TODO: developers' opinions

3.2.3 Gatsby

One rather unique framework I found - and this is a framework in a strong sense, not like the React tools above - is Gatsby. It's unique in the sense that while it's a frontend framework, it's not supposed to run in a browser. It's a part of a growing movement centered around the 'JAM stack' - "JavaScript, APIs, and Markup". That doesn't tell you much, but the main feature is that at build-time, you fetch data from your APIs, and render the application to plain HTML files, so that you don't need a server other than an S3 bucket or similar.

It's a framework targeted at a specific subset of website - not single-page applications, but more blogs or e-shops, and a workflow exemplified by Netlify. This means it doesn't need to concern itself with many features that would be missing in a frontend framework intended for a browser, and those are delegated to a different part of the stack.

TODO: opinions, structured pro/cons

3.2.4 Vue.js

TODO: Vue.js

3.2.5 Polymer

TODO: Polymer

3.3 PHP ecosystem

TODO: PHP ecosystem

3.4 Haskell ecosystem

TODO: Compare Haskell compared to the above list

- its strong points
- passable but not ideal libraries
- what's missing

3.4.1 UI toolkit

At the frontend, the UI toolkit defines what the entire application will look like, its architecture as well as the tools it can use. There is significant pressure on well-defined types in Haskell, and the UI libraries have pushed strongly in favor of FRP (Functional Reactive Programming) or its derivatives (see 'Elm architecture' [TODO: ref]).

I've managed to find five production-ready libraries that are usable with GHCJS. Out of these five, `react-flux` and `transient` are unmaintained, and `reflex`, `miso`, and `concur` all wildly differ in their philosophy, architecture, and maturity.

`Reflex` ([TODO: ref] and `reflex-dom` [TODO: ref], its DOM bindings) seems like the most actively maintained one, and also the most promising one regarding its future outlooks - it's sponsored by Obsidian Systems [TODO: ref, again] and actively developed for general use on GitHub. [TODO: describe its FRP, ...]

`Miso` [TODO: ref] - TODO: Elm architecture re-implementation in Haskell

`Concur` [TODO: ref] - TODO: an experimental architecture but actively developed, variants in PureScript as well. A technology to investigate in the future, but no ecosystem right now and not fully mature.

TODO: examples of Reflex, Miso, Concur

3.4.2 Build tools

The UI toolkit constrains the choice of possible build tools. In Haskell, there are three mature options - Cabal (new-build) [TODO: ref], Stack, and Nix. There is also a new fourth option aiming for the best of these four, Snack [TODO: ref]. While it's not yet mature enough for serious use, it's a tool worth future investigation.

Cabal - old, Cabal hell. Stack came, divided the Haskell community. Nix came out of nowhere, converted a significant portion but isn't yet dominant, and now cabal new-build is almost equivalent to stack. Free choice between cabal and stack, nix is more capable in general but slower learning curve. TODO: more about snack

GHCJS ecosystem not so well supported with Cabal (old or new one), Nix is usually recommended at the frontend (one command setup, binary cache, cross-platform), Reflex especially - uses the great Nix cross-compilation capabilities for Android, iOS, desktop. Stack usable for plain GHCJS dev, but old GHC (7.10?).

Slow compiler - common workaround is to use the REPL, but there are other solutions like Snack which are promising although not widely used.

GHCJS output size and speed - GHCJS should be obsoleted by WebAssembly very soon - it's already in alpha state, and I expect it will be ready for production by the end of 2019.

Lack of editor integration - solved with LSP+HIE (usable in VSCode and other modern editors), but Emacs is still the safest choice. Hlint (linter), Hindent (formatter) built into HIE and Intero both, same goes for 'go-to-definition' and 'type-at-point' features common to modern IDEs. What's missing is debugger integration - usually via GHCi only, but projects like haskell-dap (+ Phoityne editor plugin) exist.

3.4.3 UI components

Preferred approach - implement UI and logic inside application code. FFI is usable and quite simple for bigger components (see reflex-dom-ace).

Reflex and Concur - self-contained components and global state both (-> reusability) Miso - single state atom only (see TEA)

3.4.4 Forms

No good options exist for any of the frameworks. While there are some attempts at a forms library for Reflex, there is not a single feature-complete library. Part of the reason - validation-only libraries exist in Haskell and commonly used (see `validation`, `digestive-functors`). There are forms libraries at the backend (see `yesod-form`) with almost automatic form generation.

3.4.5 Routing

Miso has routing built-in. There are several attempts at a Reflex routing library but not a single accepted solution. Concur with its small ecosystem doesn't have even that.

3.4.6 Mobile/desktop apps

Reflex has this built-in via reflex-platform. Obelisk, building on top of reflex-platform, includes bundling apps for App/Play Store. Cross-compilation via Nix to Android/iOS, reflex-dom's bindings to WkWebView on desktop. I haven't found any attempt to do this for Miso or Concur (Concur has beginnings of React Native and SDL backends, but the development seems to have stalled).

3.4.7 Accessibility

I haven't found anything related to this topic, so I must assume no one has even attempted to tackle this topic. Building accessible applications by yourself isn't hard though.

3.4.8 Optimistic updates

I haven't found anything related to this topic. For FRP though, this should be implemented at the data or network layers though, so this is something to work towards when building the 'offline storage' library as per my assignment.

3.4.9 Web Plaftorm

The library JSaddle wraps the entirety of the Web Platform APIs using WebIDL. Any UI library can use this library, though there are limited event-based bindings, which means writing manual wrappers. (Reflex has some machinery for a subset for it, Miso has subscriptions for some of them, Concur uses ghcjs-vdom which has also some limited bindings.)

3.4.10 Prerender (Isomorphic rendering)

Reflex has explicit support, `preRender` allows even for two variants of an element if it doesn't render well. Miso has explicit support as well. I haven't found anything like this for Concur.

3.4.11 Support tools

Linter - Haskell standard is Hlint, support for custom rules, well-supported and mature. Code formatter situation is more divided, Hindent follows gofmt with a single code style for all code (but doesn't cover edge cases esp. for type-heavy code, so not ideal). Brittany is an ambitious project using GHC's parser itself, nicely designed formatting rules, but supports only a limited subset of the language. Several other projects, some unmaintained, some brand new, but Hindent seems to be standard at the moment.

The situation around code generators isn't ideal. Stack bundles several templates, but is limited to Stack users. Cabal has a single template, nothing else. Nix doesn't care about scaffolding. Several other tools, mostly unmaintained - Summoner is a notable one, with a fixed project structure; Hi is another one, supports arbitrary templates. There is no standard tool, usually only 'git clone' a project template and start developing. Obelisk has a standard structure, but it's a very limiting one (Snap as the only server library, predetermined routing library obelisk-route, ...).

TODO: more code generation tooling - 'rails new controller/scaffold/module/model/migration'

3.4.12 Quality Assurance

On this front, Haskell is very developed, at least theoretically. QuickCheck originated in Haskell and quickly propagated to many other languages - followup tools like SmallCheck and similar. That's generative, or property-based testing, best in class.

Classical unit and integration testing has also many options - hspec, hedgehog, tasty, doctest, ... Mocking via free monads or other MonadX implementations. TODO

End-to-end testing - Selenium webdriver wrappers exist, and Selenium is the standard for testing servers and clients. (Or alternately shelltestrunner for testing CLI based applications.)

Benchmarking - best in class - criterion, no competition. Weigh - allocation measurements as a benchmark.

3.4.13 Documentation

It is generally agreed that documentation is Haskell's weakest point. Despite having a great standard API docs tool (haddock), documentation is often an afterthought, with incomplete docs or unclear starting points (which means no tutorials etc. either).

3.4.14 Server-side tools

- Communication - typed APIs (HTTP - servant, WebSocket - some attempts, nothing production-ready)
- Entities - many options, some great, some less
- Migrations - weak point, I've found many half-baked implementations, but no standard solution (not only a Haskell problem, Liquibase is the only standard here, and that's SQL only)

3.5 Implementation plan

TODO: Somehow use the "Evolving frameworks" article

In this thesis, I'll focus mainly on the tools that are necessary specifically for PWA development, not general quality-of-life libraries.

Goal = app that fulfills the basic PWA criteria:

- 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 achieve this, I need to create:

- 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.
- A library or a script that will render HTML 'shells' of all pages on the site, for fast first load.
- CLI tool (requirement from assignment? "support tools")
- An offline storage library (some variant of it)

Not required by the checklist, but would improve the quality of my work:

- A template of an application, with predefined internal architecture, that uses all of the above libraries
- A utility library for querying and caching data from an API, be it HTTP, WebSocket, GraphQL, or others.

Stretch goals:

- a library to use in code shared between the server and client - a way to define the shape of the transport channel (and its API for non-Haskell applications)
- a server library, to allow user code to implement the specified protocol
- a client library with a storage component for entities and pending requests

TODO: talk about how these tools will fit into a 'platform', so that I check of a box from the assignment

An application using these tools and libraries would consist of a server written independently, and of a browser application. Implementing the communication between them is left to the developer, as is implementing many common conveniences usually provided by a framework.

Stretch: An application using these libraries would consist of a server and a client sharing code that contains the definition of their communication channel. I do not yet know how much these libraries would affect the shape/architecture of the server, but the client library would form the core of the client - with the libraries from the previous plan forming the shell.

These goals also include fully documenting the code written, as well as testing and benchmarking it to remove the most obvious bottlenecks.

TODO: talk about Serverless, JAM stack, and alternative app structures:

- Client only. An application that doesn't need to communicate with a server, like a web presentation, or a blog, a set of pre-compiled HTML+JS files.
- Server only. Either just an API, or a plain HTML website with no JavaScript.
- Server and client, with the client rendered during run-time by the server.
- Server and client, with the client rendered during build-time and served separately (e.g. via an S3 bucket).
- Server and client, with the client re-rendered on demand, whenever the data that it shows changes. This is the shape of a 'JAM stack' application.
- Projects with multiple clients and/or servers (even more pressure on supporting tools)

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

5.3 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