Comparing Haskell Web Frameworks

Beerend Lauwers Augusto Martins
Frank Wijmans
{B.Lauwers, A.PassalaquaMartins, F.S.Wijmans}@students.uu.nl
Utrecht University, The Netherlands

March 7, 2012

Abstract

This report describes the biggest problem in programming for the web, statelessness, and how features that are available in Haskell and other Functional Programming Languages can help in avoiding or mitigating those problems. We picked what we believed to be the three currently largest and most active Haskell projects for web development — Happstack, Snap and Yesod — and compared them qualitatively based on the features they implement or allow to be implemented. Furthermore, we analyze several interesting third–party libraries for web development. We also pay attention to the use of some of Haskell's advanced language features in helping make web programming easier and less error–prone. Finally, we conclude by describing each project's strong points and ideal use cases, and provide areas of improvement for each web framework.

1 Introduction

The development of dynamic web pages is a field that has been mostly dominated by dynamic programming languages. These languages tend to have a good level of expressibility, flexibility and they usually provide good (or even just "good enough") performance for this use. However, they usually lack most of the advantages many functional programming languages provide, such as type–safety and referential transparency. The use of the web has also evolved in the past decade to provide sets of dynamic web pages as a single application for end–users, following a client/server model. Given the stateless nature of HTTP as the underlying protocol this only aggravates the need for a set of good programming tools to abstract away from this limitation. In this report we present an elaboration on this issue and some solutions developed using the Clean programming language and some web frameworks for Haskell.

Haskell is a functional programming language that has been presented by some as an option for web development. [1, 2] It is a lazy and strongly typed language that provides the programmer with type safety that can catch several common programming mistakes. However, the presence of a type inference system also frees the programmer from having to extensively and explicitly declare types, reducing the amount of work required, and giving the same "feel" provided by dynamic languages such as Ruby or Python, which are popular for web development. The availability of a mature implementation in the form of the Haskell Platform also enables the language user to take advantage of the language laziness without taking too much or any of a performance hit.

From a programming point of view, the major issue with interactive applications based on the web is the stateless nature of the environment. Most developed solutions to this issue involve some sort of abstraction mechanism by storing and reloading state information on each connection. The iTask work flow management framework developed in Clean presents an alternative solution to solving the statelessness issue by storing partial computation as the state data. Section 2 presents the issue and these developments with more detail.

In general, a web framework will provide state management and I/O both from web connections and from back–end storages such as databases. Several frameworks provide other features such as automatic URL generation, type–safe HTML form generation, extensive authentication capabilities and in–memory databases. Furthermore, some frameworks provide scaffolding tools for easy installation and configuration of a website. In Section 3 we present the following frameworks for web development in Haskell:

1. Happstack
2. Snap
3. Yesod

These frameworks have been selected because of our impression that they are the most promising frameworks with the most active development communities. In Section 4, we present a more in–depth comparison of the advanced features that are present in the libraries available for the Haskell developer. We also present a comparison of the web frameworks cited above considering both the advanced features and external factors such as quality of documentation and project status.

In section 5 we present a conclusion of the comparison of frameworks and list some of the promising developments.

2 Problem

With the increasing use of the web as a platform for interactive applications, the need for better tools for developing those applications also increases. The main technique used is the abstraction away from the stateless nature of the HTTP protocol.

A web application is one or a set of web pages that are at least partially dynamically generated and may allow the user to perform actions such as input or query data stored remotely. It is usual for the user to initiate the application by loading a web page with which the user receives some session identifying information. Each action the user performs or, through client side scripts, that the browser performs on the user's behalf, can potentially require informing or querying the server. Receiving the request, the server has to identify the client session and fetch (or even calculate) the user data, possibly from external sources. This can put unnecessary strain on the server or its external sources. A common solution is to save user information along with the session identifying information. While this may impact bandwidth usage, it does mean that only user identification information must be saved server–side.

Interactive applications present another issue: that of validation. Since data presentation (on the client) and processing (on the server) are decoupled and the presentation is out of the application's control, it is also important to validate form data with caution on every request. It must be assumed that the received data could have been altered, tampered with or corrupted, and not validating data can allow issues such as disrupting non–critical services or exposing sensitive or personal data. It is also important from a practical point of view to have ways to properly validate and format outgoing data in automated ways to reduce the amount of potential programming errors passing unnoticed.

Research has shown that the type–safety provided by strongly typed languages, when used properly, can automatically detect or even eliminate some of the validation and formatting issues cited above. [3]

One example can be found in Clean, a functional programming language somewhat similar to Haskell: iData, which stands for *i*nteractive *Data*, is a toolkit developed in the Clean functional programming language.[**?**] It provides functionality to automatically generate interconnected HTML forms from *iData* elements. An *iData* element can be manipulated by both the programmer (via code) and the users (via a HTML form). The (possibly modified) serialized state of an *iData* element can be saved on disk on the server, on the HTML page itself, or in the session data. This serialized state is used to provide type–safety when data has been entered, by checking if the input value is of the same type as in the serialized state.

Built upon this system is iTask, a work flow management system written in Clean with a web front–end. In iTask, one can define work flows for processes and data [4]. Work flows can contain forms with default values for data input, data processing and storage or display of data. iTask relies and improves upon iData to provide automatic generation of type–safe work flows, even when multiple users and complex task dependencies are involved.

Apart from abstracting away from statelessness, web applications may become increasingly complex and require a large amount of resources. Web frameworks that are able to mitigate this by providing easy generation of HTML, automatic generation of URL routing and efficient use of server resources and bandwidth are then a welcoming sight. Haskell's type safety and laziness allows web frameworks written in it to perform quite well, with even the slowest Haskell-based web framework outperforming PHP–, Ruby– and Python–based web servers tenfold. [2]

3 Web Frameworks

Haskell–based web frameworks are a recent concept. HAppS was one of the first Haskell web frameworks available, and was announced publicly in 2005. Development continued until November 2008, after which Happstack forked off of HAppS. Happstack's oldest version on Hackage stems from March 2009, and Yesod's and Snap's first versions were available on Hackage on March 2010 and December 2010, respectively. All three frameworks have grown at a high rate and have become fully–featured web frameworks. Many independent web–based libraries continue to originate and be maintained, making the choice for a web framework written in Haskell all the more attractive.

3.1 Happstack

Happstack forked off of HAppS in 2009 and built upon the majority of HAppS' libraries to become a fully fledged web framework. Its main strengths are its high level of flexibility and its innovative MACID (in–memory) data storage system.

3.2 Snap

Snap is a web framework launched to the public on May 2010 centered on the creation and use of modules called Snaplets. The framework is composed of a core package with most of the common code and the API for writing Snaplets and Snaplet packages for wrapping access for individual functionality.

The top–level Snap web application initializes and parametrizes every Snaplet it will use in an initialization function. On this initialization function, sub–Snaplets are created, routes are added to the project and any other setup activity is defined and a *SnapletInit* data type is returned encapsulating a record with all the immediate sub–Snaplets' handlers (of type *Handler*). This record is usually created with the *makeLenses* Template Haskell function.

It is through the created *SnapletInit* that the Snap application is initialized and served. This function is also called automatically on reload.

3.2.1 Snaplets

Each application developed in Snap is usually composed by one or more different (sub) Snaplets but is also itself a Snaplet. This allows for modularization as each Snaplet can be treated as an isolated application/component and can also be instantiated multiple times in the same application. After being initialized, each Snaplet is used (possibly by other sub–Snaplets) through a *Handler*.

The *Handler* has a *MonadState* instance that allows the user to use the regular *get* and *set* methods from the State monad. It is worth mentioning that the states seen by all Snaplets, except for external interactions using IO (e.g. with databases or file system), are isolated per served request.

Snap has an API that focuses on allowing Snaplets to have access to environment and state data. Most code that do not depend on either can be left outside any Snaplet (and, in fact, be used by any other Haskell web–framework).

3.3 Yesod

Development on Yesod started in 2009. It has quickly grown into a mature web framework featuring tightly integrated web framework capabilities. Yesod is the only web framework in the comparison that natively supports internationalization and the only one that started without an in–house web server component. This changed in 2011, when development on the *warp* web server package commenced.

4 Analysis

Most frameworks present a subset of common features that aid the developer in solving the most common problems for developing web pages. These problems range from dealing with path routing and serving files to handling session information. Not every framework implements every feature, opting instead to use external, stand–alone libraries. We first review the frameworks' implementation of common features in Section 4.1 before giving a general analysis of each web framework in Section 4.2.

4.1 Feature analysis

4.1.1 Path Routing

When serving web–pages from complex projects, the layout of the files on the project root do not necessarily map to the served URLs on a one–to–one basis. Some URLs may even be dynamic, relying on some form of server state, query string or following complex pattern rules being processed after each request. In general, techniques to facilitate returning the correct HTTP responses can be grouped under the label "Path Routing".

The path routing system in Happstack uses the *ServerMonad* and *MonadPlus* classes to allow for easy and clear creation of routes using guards. *ServerMonad* is a specialized version the *MonadReader* class, and provides functions for modifying and reading an HTTP *Request*. Static elements are checked with the *dir* and *dirs* guards.

Parts of the URL can be extracted using the *FromReqURI* class: the *path* guard will capture a string, which can then be used in the generation of the response: One can create a *FromReqURI* instance for an arbitrary Haskell data type, so that a value of the data type is returned in place of a string.

HTTP request methods (GET, POST, HEAD, etc.) can also be used to guard against, allowing for custom responses based on the request method. Through use of the *MatchMethod* class, it is possible to match on several methods at once or provide a custom method matching function.

It is also possible to guard against the *host* header to allow multiple domains to be hosted on a single Happstack server. Finally, guarding against the HTTP request itself is also possible using *guardRq*.

In Snap, every Snaplet allows for the adding of routes relative to its own instantiation. Those routes can point to sub–Snaplets or directories to be served directly. On startup, the Snap core takes care of unifying the routes added in every instantiation into a single set of valid URLs to be served. There is not, however, any sort of validation of the routes being added other than the verification of the correct instantiation of the sub–Snaplets.

Yesod first splits up a requested path into several pieces, splitting the URL at all forward slashes. It will then attempt to match this list of split elements to the site map, and, if successful, call the appropriate *Handler* function.

For example, a simple site map could look like this:

```
/person/#String  PersonR  GET POST
/year/#Int       YearR    GET POST DELETE
/page/faq        FaqR
```

The resulting data type would look something like this:

```
data MyRoute = PersonR String
             | YearR Int
             | FaqR
```

Handling requests to these URLs would then require, at the very least, three handlers: *handleFaqR*, *handleYearR* and *handlePersonR*. However, one may wish to call a different handler depending on the request method. In the case of *YearR*, the handlers would then be *getYearR*, *postYearR* and *deleteYearR*.

Like Happstack, Yesod allows for extracting a part of the URL and passing it to the *Handler*. The *Handler* monad is a monad transformer stack consisting of a *Reader* providing access to information about the request, a *Writer* for adding extra response headers, a *State* for session variables, and a *MEither* transformer for handling static content and sending error responses.

Similar to Happstack's *WebMonad*, Yesod offers "short–circuiting" behaviour with the *MEither* monad. The current computation is escaped and a response is returned to the user.

Besides built–in options, there is also the *web – routes* package, which allows for highly type–safe URL routing. It is designed to be highly flexible: mapping URL types to strings and vice versa can be done via Quasi–quotation, Template Haskell, generic programming libraries, parser libraries or custom functions. A data type is defined that represents all valid routes. Then, a *PathInfo* instance is derived using TH:

```
data SiteMap = Index | Page Int deriving
     deriving (Eq, Ord, Read, Show, Data,
     Typeable)
$(derivePathInfo ''SiteMap)
```

The library's *RouteT* monad transformer is wrapped around the server monad. In the example code, we will use Happstack's server monad: *ServerPartTIO*. *RouteT* is a *Reader* monad that guards against incorrect URLs, because *RouteT* is parametrized by the URL data type *SiteMap*. The *RouteT* also provides the *ShowURL* class, which provides the *showURL* (URL data type to *String*) and *showURLParams* (hostname, port, path prefix) functions.

After deriving the *PathInfo* instance, routing a path is trivial:

```
route :: SiteMap → RouteT SiteMap (ServerPartT
     IO) Response
route url =
    case url of
      Index      → mainPage
      (Page id) → selectPage id
```

where *mainPage* and *selectPage* are Happstack *Response* generators.

An interesting extension to $web-routes$ is the $web-routes-boomerang$ library. This library allows one to define a single grammar that generates a parser and a printer for URLs. Greater control can then be exerted over the appearance of URLs while retaining maintainability.

4.1.2 Serving Files

Customized file and directory serving can make static text files look like HTML pages and provide custom layouts to directory indices. Both Happstack and Snap provide granular functions for creating custom functions.

Happstack provides high flexibility for serving static files from disk. It uses its *WebMonad*, *ServerMonad* and *FilterMonad* to do so. The *FilterMonad* monad is used for manipulating *Response* values, such as setting response codes and content types. Creating your own response–generating function is also possible. The *WebMonad* monad is also quite nice: it allows you to escape the current computation and immediately return a *Response*. This *Response* can have have a different content type from the original *Response* that was being built.

In order to serve a directory, one uses the guards described in Section 4.1.1 together with the function *serveDirectory*. Rolling your own *serveDirectory* or *serveFile* function is possible by piecing together functions from the *Happstack.Server.FileServe.BuildingBlocks* module.

Yesod relies on the web server for serving static files from disk. It does provide some functions for looking up files and serving them, but creating your own functions will be harder.

Snap provides a utility module, *Snap.Util.FileServe*, with many interesting functions, such as automatic and customized generation of directory index files and dynamic MIME–type handlers, which allow for things such as pretty–printing source code automatically.

4.1.3 HTML Generation

Some web frameworks originally developed their own HTML library: Yesod developed Hamlet; Snap developed Heist. Currently, these libraries and several others can be used in every web framework with little to no extra code required.

Happstack ships with BlazeHtml and HSP (Haskell Server Pages), but also has boilerplate modules for Heist, Hamlet and HStringTemplate. Snap uses Heist as its default HTML library, and Hamlet is still the default for Yesod.

An overview of the most popular libraries:

1. BlazeHtml is a fast, combinator–based library that allows you to mix HTML tags with your Haskell code.

2. Heist is an XML template engine that separates the XML templates from the Haskell code, improving maintainability.

3. Hamlet is an HTML library that uses Quasi–Quotation to use HTML tags directly in the code. It is type–safe and helps to prevent XSS issues and 404s.

4. HSP gives you the ability to embed XML tags inside your Haskell code, and can thus also generate XML code.

5. HStringTemplate is based on Java's StringTemplate library, and allows for both embedding templates inside Haskell code as well as loading them from disk.

While on–disk templates allow for fast changes, the speed of compiled code is lost. The *happstack––plugins* package mitigates this by providing a system for automatic type–checking, recompilation and reloading of modules on a running server.

4.1.4 JavaScript And CSS Generation

The same techniques for HTML generation could also be used for JavaScript and CSS generation.

Yesod provides the *Cassius*, *Lucius* and *Julius* quasi-quoters. *Cassius* allows insertion of variables into CSS code, such as RGB values and URL's generated by Haskell. It uses whitespace rules instead of braces and semicolons. *Lucius* uses the traditional CSS syntax, but retains *Cassius*' insertion properties. It also allow one to nest CSS blocks. For example, here is a normal CSS file:

```
.article p { text-indent: 2em; }
.article a { text-decoration: none; }
```

Compare this with the following:

```
.article {
    p { text-indent: 2em; }
    a { text-decoration: none; }
}
```

Finally, *Julius* is a very simple JavaScript generator that, like *Cassius* and *Lucius*, allows insertion of variables and URL's. However, it also supports insertion of templates.

Happstack provides a boilerplate module for *JMacro*, an indepedent library that is more ambitious than Yesod's *Julius*. Happstack does not have a CSS generator.

JMacro provides JavaScript syntax checking, insertion of Haskell values and use of Haskell techniques such as lambda expressions:

```
// Functional style declarations:

// Cannot be partially applied
fun add x y → x + y;
// Can/Must be partially applied
fun addTwo x → \y -> x + y;

// Traditional JavaScript
var addThenMult = mult( 2, add(2,3) );
// Haskell-styled function application
var multThenAdd = add 2 (mult 2 3);
```

It also provides automatic variable scoping: scoped variables will be renamed in the final JavaScript to prevent overlap with variables of the same name declared elsewhere. Antiquotation allows one to use Haskell code directly within JavaScript code:

```
let foo = 5 in renderJs [$jmacro|var x = `(foo)
    `;|]

//Result
var jmId_0;
jmId_0 = 5;
```

Snap does not have CSS or JavaScript generators, and does not appear to have a Snaplet for *JMacro*.

4.1.5 Parsing Request Data And Form Generation

URLs with query strings, GET and POST requests and cookies contain information that has to be extracted.

Happstack provides the *HasRqData* monad to easily extract information from the query string, GET and POST submissions and cookies. Its *look* function searches through the query string and form submissions, while *lookCookie* does the same for cookies. In order to extract data from form submissions, the body must be decoded with *decodeBody*, after which it is temporarily saved in a flat file, which is garbage collected afterwards.

The *lookRead* function uses the type system to parse a parameter into a Haskell value, and integrates seamlessly into the *RqData* error handling for when parsing fails. A custom parsing function can be supplied with the function *checkRq*, which also allows for setting custom error messages. The function also allows you to enforce a specific subset of input (for example, only allowing Haskell *Int*s between 1 and 10). Optional parameters are provided with the *optional* function from *Control.Applicative*.

Yesod uses the $yesod - form$ package to generate HTML forms. The library integrates seamlessly into Yesod. It is quite elaborate, featuring type–safe form generation, parsing submitted form data into Haskell data types, JavaScript validation code generation, and an applicative way of combining forms (*AForms*) to create more complex ones. Moreover, form generation can also be done monadically (*MForms*), which gives the programmer complete control over the form's layout. Finally, it is also possibly to have pure input forms (*IForms*) which only parse data, but do not generate HTML (perhaps because the HTML code has been written by hand). Note that an *IForm* does not support client–side validation or error reporting.

The *DigestiveFunctors* library supports Happstack and Snap and is used to create composable HTML forms. Yesod's form library used some concepts from the *digestive − functors* library.[?] The library is inspired by the *Formlets* library.

Simple form elements or "formlets" can be defined in terms of primitive input tags, after which more complex forms can be composed using the formlets. An example would be a personal information form consisting of three fields:

```
data Info = Person String String Int deriving (
    Show)
infoForm = Person <$> inputText Nothing <*>
    inputText Nothing <*> inputTextRead "Could␣
    not␣parse␣value" (Just 25)
```

Creating a form with three of these forms is then as simple as:

```
data Group = Persons Info Info Info deriving (
    Show)
groupForm = Persons <$> infoForm <*> infoForm
    <*> infoForm
```

The *FormState* newtype is a *State* monad in which the "formlets" will be composed, and provides a unique identifier for each $< input >$ tag, which is used for field–specific error reporting.

4.1.6 Session Management

Session management is a common way to uphold user state. Yesod's library *clientsession* is used by Snap to implement session management. Happstack does not have an official package available, but there are a few external libraries available.

The majority of the libraries do not support saving values other than user name, password and expiry time in the session data.

Yesod's *clientsession* stores session data in an encrypted client–side cookie, which is sent along with each request. They justify this overhead by noting that no server–side database communication is required, which means state servers are not required: adding new web servers poses no issues. Aside from login information, *clientsession* provides a message system for a common use case: after a POST request, the user is redirected to a new page and simultaneously sent a success message. The library also integrates with *yesod − auth*'s ultimate destination design: when a user wishes to access a page and is requested to log in, he is redirected to his desired page after logging in.

Snap's *snap − auth* package uses *clientsession* as a base, but adds the ability to save extra data using the *CookieSession* Snap extension. At the time of writing, this package was not yet available on Hackage. The *mysnapsession* package contains a simplified version of *snap − auth*, but also supports in–memory sessions. It also provides a continuation–based programming model suited to multiple–request stateful interactions using in–memory sessions.

For Happstack, there exists the *happstack − auth* and *happstack − extra* packages. The first library supports simple sessions (login, logout, expiry) that are hashed with SHA512 and checked with a fingerprint (IP and user–agent). The second provides many handy features, including session management. Only the session ID is saved in a cookie. Other session data is saved in–memory.

4.1.7 Authentication

Related to session management, user authentication is also a staple of web development. Integration with OpenID, Facebook Connect and so on are commonly requested features. The *authenticate* package, spun off from Yesod, forms the basis for both Yesod's and Happstack's authentication libraries.

The *authenticate* package supports BrowserId, Facebook Connect, Kerberos, OAuth, OpenID and rpxnow. *yesod − auth* adds E–mail, Google Mail with OpenID and HashDB to that list. *happstack − authenticate* adds support for multiple authentication methods per account, as well as multiple personalities per user account and themeable BlazeHtml–based templates. Snap's *snap − auth* provides only basic user/password authentication functionality.

4.1.8 Persistence

Similarly to HTML generation, the developers of two of the frameworks created their own back–end for persistent data storage: *happstack − state* for Happstack and *persistent* library for Yesod. *happstack − state* has been succeeded by *acid − state* and is now independent of Happstack, and is thus discussed at the bottom of this section. It is still the recommended persistence package for Happstack. Both *acid − state* and *persistent* are non–relational.

The *persistent* library allows one to store arbitrary Haskell values. It does so by generating boilerplate code using a combination of Template Haskell and Quasi–quotation:

```
mkPersist sqlSettings [persist|
Person
    name String
    age Int
|]
```

This code block will generate a representation that can be explained to a database backend (supported out of the box are MongoDB, PostgreSQL and SQLite), mapping the data type's parameters to *PersistValue*s, which correlate to a single field in a database column. These *PersistValues* are marshalled to a *PersistField* class instance, which can be correlated to a database column. An entire database table is viewed as a *PersistEntity* class instance. Both classes make use of phantom types, whereas previously Template Haskell was used. Finally, a primary key data type is generated for the *Person* data type: *PersonId*.

Data migrations caused by database schema changes are mitigated using several helper functions available in *persistent*. Data type changes and additions of new fields and entities are done automatically. Field and entity renames and data removal are not automatic. The *printMigration* function will print all actions the migration system will do without actually performing them.

Insert, update and deletion queries can be written in Haskell themselves, and *persistent* provides several helper functions. For example, this code block will create a query that returns all persons with an age of 22 in ascending order:

```
persons ← selectList [Age ==. 22] [Asc Age]
```

Because every entity has its own primary key data type, extra type safety is achieved. For example, a query that attempts to use a *PersonId* to select a *Car* value will result in a compile–time error, as a *CarId* would be required.

While defining a data type for use in a database, it is possible to specify uniqueness constraints, default values, nullability, foreign keys and custom table and column names.

Snap does not have an in–house persistence system. Snaplets are available for the *HDBC* and *mongoDB* packages, as well as the *acid − state* library mentioned below.

AcidState is a framework–independent package that supplies a MACID store: an in–Memory ACID data store that is capable of storing arbitrary Haskell values. It does so by writing to file which functions were executed along with their parameters. Thus, recreating state simply means re–running the saved logs. Its predecessor was *Happstack − state*, which functions similarly, but which is only available for Happstack. *AcidState* is platform–independent. User error cannot lead to corruption of data: exceptions will abort a transaction, deleted or renamed functions will be caught during re–running, and the format of the state is checked for parsability. *AcidState* makes heavy use of *MVar*s to ensure ACIDity, and uses *IORef*s for remote communication. Use of the *SafeCopy* package allows for easy future extension of data structures saved in the MACID store, as well as serialization of the data using *SafeCopy*'s *deriveSafeCopy*.

Updating the data store is done via the *Update* monad, which can be seen as the *State* monad for use in an ACID–environment. In the same way, the *Query* monad can be seen as the *Reader*'s ACID–equivalent. The TH–function *makeAcidic* will generate *SafeCopy* instances for the update and query functions. It will also generate data types based on the names of the passed functions, which are used to ensure that updates and queries incompatible with the MACID store will result in a type checking error.

4.1.9 Project management

For starting new projects both Yesod and Snap provide useful programs. Yesod provides a scaffolding tool that asks the user a few questions, after which it will generate a cabal package containing a skeleton Yesod project. Snap has a few commands to initialize a bare–bones skeleton or a simple, fully working "Hello World!" example. Happstack does not provide any tools to set up a server project.

4.2 Framework analysis

4.2.1 Happstack

Happstack profiles itself as a flexible and fast web framework, attempting to give the developer free reign over as many aspects as possible. Small boilerplate modules allow Happstack to interface with external libraries without forcing a developer to use a specific library. In–house features are designed to be as modular as possible to allow developers to roll their own functions. Happstack provides an extensive "Crash Course" [?] that will help a developer learn Happstack by playing with the code. The Crash Course goes to great lengths to explain Happstack's capabilities without introducing abstract or complex concepts. Of course, more extensive documentation can be found in the framework's API. Overall, Happstack is a good choice for the developer with a hacking mentality: checking to see what works well together and having the ability to swap out parts they don't like. Alternatively, Happstack also functions reliably as an actual website, supporting the full spectrum of possibilities of a modern web framework.

4.2.2 Snap

Snap is a framework focused on the management of Snaplets and, as such, has a relatively simple feature set. However, given its simplicity, it is also straightforward to wrap standalone libraries into a Snap project. Snaplets allow for high levels of modularization as each Snaplet can be treated as an isolated application/component and can also be instantiated multiple times in the same application. Its strength seems to be in the simplicity and solid base, and as of now, it is mostly a mid–level framework, providing the basics for a web server and extending functionality via its Snaplet concept, which can be thought of as similar to Java Servlets. The web framework comes with a few Snaplets, and several others are available via their website or via Hackage. Snap provides documentation for installing and using Snap as a simple web framework, as well as information about Snaplet creation and design. The framework's API is well documented. Snap is of interest to developers who require a small and fast web server with a minimal footprint, while retaining the ability to extend it with features they require. While no examples could be found of Snap being used as an embedded HTTP server, it appears to be a prime candidate for this role.

4.2.3 Yesod

Yesod achieves flexibility and high speed in a different manner from Happstack. This framework uses strongly integrated libraries to ensure high levels of type safety. While most of its libraries are usable in other web frameworks, the majority of Yesod's features were developed in–house. It leverages this opportunity to provide a tightly integrated web framework that helps the developer to exploit Haskell's type system to ensure type–safe URL creation, HTML and form generation, database communication, and much more. Similar to Happstack's Crash Course, Yesod provides a "Web Framework Book" [7] that explains Yesod's inner workings in detail without elaborating upon the advanced Haskell features it uses. The book's appendices goes into deeper detail into some of the designs Yesod employs, and the framework's API is also richly documented. It is only recently that Yesod has received an actual web server component. Before this, it used web servers based on the *wai* package. Its *warp* package, also built upon the *wai* package, appears to be very fast, with preliminary results showing it to be more than twice as fast compared to Snap and Happstack. [2] Yesod's capabilities will interest developers who are interested in highly performant, highly type–safe applications by leveraging Haskell's strong points.

5 Conclusion

Developing web pages and web–based applications requires the programmer to deal with the stateless nature of the HTTP protocol. Abstracting away from this lack of state by using web frameworks is the most common solution. In this paper, we presented three of the most promising frameworks for developing web pages in Haskell: Happstack, Snap and Yesod. We conclude that all three contain different development philosophies:

1. Happstack allows the programmer to choose among a very diverse set of tools while providing a good feature set out of the box.

2. Snap provides a minimal system and does not impose any limitations, but also requires the programmer to resort to external libraries for most of the required functionality.

3. Lastly, Yesod provides most of the usual functionality and tries to do so consistently, but this also makes it more difficult to divert from the given programming style.

In general, though, most of the functionality implemented in the frameworks is independent and can be ported to one another.

Because of the interchangeable nature of most of the features, we paid special attention to the specific implementation details of the main libraries. Specifically, we noted that Haskell allows avoiding common web programming mistakes by implementing libraries that take advantage of the language's type–safety features. Language extensions and techniques, such as monad transformers, type families, Template Haskell and Quasi-quotation make the programmer's life easier by reducing the probability of security holes or incomplete implementations and help with code reuse.

All the frameworks have quite active development communities, and provide some healthy competition and mutual aid by code and library sharing. However, some framework–specific issues could use attention. For instance, Happstack could make good use of a project scaffolding and management tool such as the ones found in both Yesod and Snap. More Snaplets available on Hackage would make Snap far more useful and interesting for start–up projects that may require new or different functionality over time. Snap's $snap - auth$ package could benefit from using the *authenticate* library to allow for more authentication methods. At the moment, Yesod relies on the web server's implementation for static file serving. More fine–grained control over this feature in its *warp* web server should be on its to–do list.

While there are many libraries available for HTML generation, libraries for CSS and JavaScript generation are rare. Happstack provides only a $JMacro$ library wrapper, and Snap does not provide any CSS or JavaScript generation Snaplets. Snap and Happstack could also profit from Yesod's and $web-routes$' approach to type–safe URL routing.

Because Snap profiles itself as a "simple and fast" web framework with a "minimal HTTP API", it seems to be an ideal candidate for a Haskell–based embedded HTTP server.

Finally, Happstack's $acid - state$ library, an in–memory ACID data store, provides an interesting alternative to traditional back–ends. Comparing this library's performance against more conventional SQL and NoSQL databases may prove to be very interesting.

It is also worth noting that several benchmarks show the frameworks to be quite fast. [1, 2] However, some of the benchmarks seem to be contradictory and are often relatively shallow. An extensive performance comparison can provide a more definite answer to the question of which web framework excels in what areas.

6 Acknowledgements

We would like to thank Doaitse Swierstra for his guidance during development of this report.

References

[1] Gregory Collins and Doug Beardsley. The snap framework: A web toolkit for haskell. *IEEE Internet Computing*, 15:84–87, January 2011. ISSN 1089-7801. doi: http://dx.doi.org/10.1109/MIC.2011.21. URL `http://dx.doi.org/10.1109/MIC.2011.21`.

[2] Michael Snoyman. Warp: A haskell web server. *IEEE Internet Computing*, 15:81–85, May 2011. ISSN 1089-7801. doi: http://dx.doi.org/10.1109/MIC.2011.70. URL `http://dx.doi.org/10.1109/MIC.2011.70`.

[3] William Robertson and Giovanni Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 283–298, Berkeley, CA, USA, 2009. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855768.1855786`.

[4] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An introduction to itask: Defining interactive work flows for the web. 2008. URL `http://www.st.cs.ru.nl/papers/2008/plar08-iTasks_CEFP2007_Revised.pdf`.

[5] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. 2011. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/remote.pdf`.

[6] Michael Snoyman. Yesod web framework blog. URL `http://www.yesodweb.com/blog/`.

[7] Michael Snoyman. Yesod web framework book. 2011. URL `http://www.yesodweb.com/book/`.