Master Thesis

A Haskell Web-Application for Data-Mining Competition-Results from StarExec

A thesis submitted to attain the degree of
Master of Science at the Leipzig University of Applied Sciences
(M.Sc. HTWK Leipzig)

presented by

Stefan von der Krone

B.Eng. in Media Technology, University of applied Sciences Mittweida

born on 19th of April, 1984

citizen of Germany

accepted on the recommendation of Prof. Dr. Johannes Waldmann

2014

Table of Contents

1

Introduction

This is a guide to the Star-Exec-Presenter, a web-based visualization and data-mining software developed in preparation of this thesis. The Star-Exec-Presenter is intended to be an efficient to use software for members of the termination community of the StarExec service, as well as other communities or users of StarExec. The web application features essential functionalities such as starting competitions, loading results and other data from StarExec or visualizing and filtering such data. Contributers of solvers should be able to track their progress of development of their respective solver through time by comparing the results over time.

Termination is a branch of theoretical computer science where programs such as algorithms are examined, whether they terminate, that is whether they complete. In terms of StarExec, solvers are running over a bunch of problems returning an answer for each problem: Yes, No or Maybe.

The Termination Competition is an annual event of the termination community of StarExec, where solvers from different contributors compete in several categories. In 2014 the competition was organized by my adviser, Prof. Johannes Waldman, and me. Our goal was to do both, starting the competition run as well as to present it via an automatically updated web-interface.

The 2014 competition caused a total amount of 13 Gigabyte of data and needed about 90 days of CPU time on all of the 192 Quad-Core CPUs of the StarExec

cluster. (Waldmann 2014) It it over eleven hours to complete all jobs.

Our additional goal was to be able to compare these results with those from previous ones. So we added an option to import the data from the 2007 competition hosted by the Laboratory of Computer Science at Université Paris-Sud and the following competitions hosted by the University of Innsbruck.

"StarExec is a cross community logic solving service developed at the University of Iowa (…)." (StarExec 2013) It is a technical infrastructure providing the service to run logic solvers on a powerful cluster of CPUs. It also provides an extensive web-based user interface to upload and run solvers and problems, referred to as benchmarks. A REST-based[1] API[2] is utalized by the Star-Exec-Presenter.

The Star-Exec-Presenter is the software developed as the concrete work for this thesis. It is a web application programmed in Haskel[3] with the utalization of the Yesod Web Framework[4].

The motivation of this thesis is to provide a tool for StarExec users for further research as well as some kind of standardization of hosting and running future termination competition and competitions of other StarExec communities. The goal for Star-Exec-Presenter is to be a tool, that is easy to use as well as easy to install and run. It is open source, so it can be forked und changed to better meet future needs.

[1] Representational state transfer, a paradigm to implement a server-client communication

[2] Application programming interface, a set of components (methods, protocols, tools) for using a software

[3] a functional, non-strict, declarative programming language (http://www.haskell.org/)

[4] a REST-based web application framework (http://www.yesodweb.com/)

2

Termination Competition

In this chapter I will discuss the terms of Termination and Termination of Term Rewriting. I also will describe the Termination Competition, as its 2014 incarnation was one of the goals towards the Star-Exec-Presenter was developed.

2.1 Termination of Term Rewriting

(Zantema 2000)

2.2 Termination Community

2.3 Termination Competition

(Marché and Zantema 2007)

2.4 TPDB - Termination Problem Database

3

StarExec

StarExec is the service that is utilized by the Termination Community and other communities to run their research as well as the latest Termination Competition. This chapter will examine the interfaces from which the relevant data is fetched as well as their special characteristics. I will go into the details of how StarExec works and how to get the data.

3.1 History

StarExec is *"a public web-based service built to facilitate the experimental evolution of logic solvers, broadly understood as automated tools based on formal reasoning. (…) The service, (…), is designed to provide a single piece of storage and computing infrastructure to logic solving communities and their members. It aims at reducing duplication of effort and resources as well as enabling individual researchers or groups with no access to comparable infrastructure."* (Stump, Sutcliffe, and Tinelli 2014) The work on StarExec started in 2012 as a joint development of the University of Iowa and the University of Miami. It is a cluster of 193 quad-core CPUs and 128 to 256 Gigabyte main memory each running a Red Hat Enterprise Linux.

The first public event was held in summer 2013. One year later, the Termination Competition 2014 was realized with StarExec.

3.2 Data-Model

The data-model has three major entities: the jobs, the solvers and the benchmarks. A benchmark is a formular[1] regarding a formal problem which will be processed by a solver. A solver is a programm that takes a benchmark, analyses it and gives an output regarding the given task. For example, in the case of the Termination Competition, this output should mainly consist of one of the following results: YES, NO, MAYBE. Whereby the answer YES can also contain additional information for the complexity of the problem. A post-processor is responsible for filtering this info out of the solvers log.

A job is a complex task with a set of solver and benchmarks, where each solver processes each benchmark. Each relation between a certain solver and benchmark is called job-pair. Each job-pair consists of the solver's log (stdout and stderr). The job-pair is directly connected to the result info, which contains the solver's result and additional info such as cpu- and wallclock-time, the configuration and identifying information of the certain solver and benchmark.

So, when a job is started, each solver will be started as well with a benchmark as input. The StarExec infrastructure itself takes care that all available CPUs are busy. Of course, not all job-pairs are started at the same time. If a solver finished its task a post-processor is working on the solver's output to extract the actual result (YES, NO, MAYBE). All information will then be stored in the MySQL database.

Hierarchically, StarExec is structured into communities which use spaces for further subdivision. Currently, there are several communities like "Termination" or "SMT". Each community consists of several spaces, some being owned by a user, some being public. Each space can also have subspaces. A space consists of a bunch of jobs, solvers an benchmarks. The job-results and job-pairs can be accessed through the particular job. solvers and benchmarks can be used in other spaces by copying or linking them.

3.3 Interfaces

StarExec has an extensive interface to communicate with. It provides a complex user interface via the browser as well as a RESTlike.

[1] see https://wiki.uiowa.edu/display/stardev/Project+Overview

3.3.1 GUI

The browser based web interface of StarExec is characterized by the hierarchical nature of the spaces. A user has to have access to a space to start a job as well as upload a solver or benchmark. Each space has an overview of its job, solvers, benchmarks, its subspaces and even the enlisted users. All entities listed in this overview are owned by the particular space, so the entities of a subspace can only be seen by navigating to the certain space.

As addressed before solvers and benchmarks can be copied or linked from other spaces but also be uploaded into the space. The available actions may alter according to how much access a user may have regarding a certain space. Some space may even be hidden from the user. Aside from starting jobs and other actions mentioned before, it is also possible to download certain information such as all jobs and benchmarks or a XML based representation of the space itself.

The web interface relies heavily on AJAX-request[2] to prevent each page from being reloaded on each action. So, when the user is navigating the tree of spaces, there is no actual reload of the whole page but an asynchronous fetching and updating of the page's content. This behavior is also applied to the handling of a job's progress. So, until a job is finished the web page automatically keeps track of the progress by regularly sending AJAX-requests to StarExec.

3.3.2 API

Another part of the interface to StarExec is also a part of the GUI but only within download-links or forms: a RESTlike API. This interface is heavily used by the java tool StarExecCommand[3] which is a commandline tool for the communication with StarExec. It provides a shell-like prompt that can be used to do certain action on StarExec. StarExecCommand was also some kind of inspiration for the design of Star-Exec-Presenter when it comes to the interface to StarExec.

In the following I want to list the important URLs that are also used by the Star-Exec-Presenter:

`https://www.starexec.org/starexec/secure/j_security_check`

[2] Asynchronous JavaScript And XML, a technique to dynamically load data remotely and update the content

[3] see https://www.starexec.org/starexec/public/starexeccommand.jsp

Of course, without this URL StarExec is actually of no use, because a user has to be logged in. With the correct credentials a POST request has to be send to StarExec. If StarExec returns a valid session ID the user is successfully logged in. Star-Exec-Presenter itself doesn't require the client to provide his own StarExec credentials, as the application uses its own. So, everyone who starts our web application has to provide valid login credentials.

`https://www.starexec.org/starexec/secure/download`

Another important URL is the one to download certain data. This URL is used consistently throughout all download links. It requires GET parameters attached to it, the most important being the `id` and the `type`. The `type` defines the type of entity such as `job` or `spaceXML`. The `id` has to be a valid identifier of the entity. If `type` is set to `job` the job-results will be downloaded in the form of a zipped CSV. Alternatively, if `type` is set to `spaceXML` a zipped XML containing the space's hierarchy will be downloaded.

`https://www.starexec.org/starexec/services/jobs/pairs/{pairId}/stdout`
`https://www.starexec.org/starexec/services/jobs/pairs/{pairId}/log`

Both listed URLs are related to the job-results. As the job-results consists only of some meta-data as well as the final result, the actual output has to be downloaded via these two URLs. Conveniently, the response is plain text. The placeholder `{pairId}` refers to the identifier of a job-result.

`https://www.starexec.org/starexec/secure/details/job.jsp`
`https://www.starexec.org/starexec/secure/details/solver.jsp`
`https://www.starexec.org/starexec/secure/details/benchmark.jsp`
`https://www.starexec.org/starexec/secure/edit/processor.jsp`

To receive additional infos of the certain entities Star-Exec-Presenter works with, there are four additional URLs to observe. As StarExec has no way to get these information other than via web pages, Star-Exec-Presenter parses the required data from HTML. Each URL requires a GET parameter `id` set to the particular entity's identifier attached to the end.

4

Use Cases

In this chapter I will list the use cases which the Star-Exec-Presenter is mapped onto. These use cases range from simple tasks, for instance displaying solver-results from StarExec, up to more complex ones, like starting and monitoring a competition.

4.1 Start a predefined Competition

One of the core use cases of the Star-Exec-Presenter is the ability to start a competition run on StarExec. As StarExec actually doesn't know what a competition in our terms is, Star-Exec-Presenter should know it. So all meta-information of a competition should be managed by the Star-Exec-Presenter. These meta information include hierarchical information of the competition's organisation as well as the details which we'll send to or get from StarExec.

Hierarchical information is meant as how each solver and benchmark is organised within the competition. Each competition consists of several meta categories which cover a general topic in the competition. The 2014 Termination Competition has the following: "Termination of Term Rewriting (and Transition Systems)", "Complexity Analysis of Term Rewriting" and "Termination of Programming Languages".

Each meta category is devided into smaller, more specific categories, each being a representation of an actual job on StarExec. These categories may be best described

by the meta categories "Termination of Programming Languages", as there is a category about the programming language C. Other categories could bring up Java, Logic Programming or Functional Programming.

4.2 Displaying Results and Outputs

The core function of StarExec is not only to run the solver on specific benchmarks but also to host the results and outputs of each run. So a major task for the Star-Exec-Presenter is to display that data in an appropriate way. The explicit data is the result of a solver with a specific benchmark and its CPU- as well as Wallclock-Time. Additionally the solver's output is important which is accessible on StarExec with the job-pair, a unique combination within the specific job of solver and benchmark. The output can contain plain text or even XML or HTML.

4.2.1 Displaying a Competition and its Results

The Star-Exec-Presenter is a tool that ran during the 2014 Termination Competition. It should display the current status of the competition run with a compact web interface that automatically updates itself.

4.2.2 Interpret Outputs as HTML or XML

Some solver explicitly log an HTML- or XML-proof with each run in its output. This proof should be detected as well as extracted from the output. Also the respective XSL-File for the XML output should be used to display the proof in the browser.

4.3 Compare Results, Querriyng

Simply displaying the results and outputs of each solver-run is not enough. It is more important to compare the results between several solver. Are there differences between their results on specific benchmarks? Also, a list of jobs should be displayable at once. A historical comparison is the most important aspect of this use case, as the developers of a solver would want to track the evolution of their tool.

As important comparing the result is as overwhelming is their amount, so a beneficial use case is querriyng the results according to specific filters. Such filters can

eliminate unimportant results from the representation. For instance a benchmark is uninteresting if all solvers have the same result, so all those benchmarks can be filtered out.

The user should be able to concentrate on data which meets his or her interests. This data could tell which benchmark causes incorrect results with the own solver. Also it can illustrate changes in the solvers results in comparison to past runs, fo instance a once solved benchmark could now be unsolvable. This is the data we want to querry in terms of data mining.

5

Technical Requirements

This chapter discusses the requirements that were agreed to. Which data is interesting and how is it handled? How should the application be build, which kind of data should be accessible through the Star-Exec-Presenter? And how should the application incorporate with StarExec? All requirements are infered from the previous chapter.

5.1 Solver and Post processor output

To display the results of a job on StarExec we must now which data actually is important. So, basically StarExec knows two different kinds of resulting data. The first is a table which contains the resulting data of each solver-benchmark combination (the job-pair) of the job. It tells us the final result of a job-pair as well as its CPU- or Wallclock-Time and its current status. Each job-pair has a unique identifier which leads us to the second kind of resulting data: the output of a job-pair which is accessible separately. It contains the standard-out of the solver and a log generated by StarExec.

The aforesaid table should be displayed wellarranged with the Star-Exec-Presenter. Important data kinds are of course the result of the job-pair as well as the CPU- and Wallclock-Time. The presentation should be easy to read and its information

gathered quickly.

The output of the job-pair should be scanned for XML to determine wether it contains a proof. The corresponding presentation should give the chance to display that proof according to its required XSL-Stylesheet.

All data should be persistently stored in a Postgre-SQL database.

5.2 Compact web interface

The web interface of StarExec is quite verbose and not specifically matching the needs of members of the termination community. So, the Star-Exec-Presenter should feature a much more compact web interface which clearly guides the user to the destined information. Readability is important.

5.3 Short response time

As StarExec is located in Iowa and some requests require expensive fetching from its database some responses may need up the seconds to be fully available. This is not desired as some users may consider it an error of the Star-Exec-Presenter. So our tool should cache all data that it loads from StarExec. An additional information should be shown when such data is about to be fetched. The desired effect is that each request is responded as early as possible. Also every additional calculation which is repeatedly needed should be cached, too.

5.4 RESTful API

5.5 Import old data

6 Design

The Star-Exec-Presenter is actually the concrete work of this thesis. As a REST-based web application it is an interface to the StarExec service and it provides a caching mechanism. This chapter will give detailed insights in the architectural approach to build the Star-Exec-Presenter according to the requirements.

6.1 Application Structure

Star-Exec-Presenter is written entirely in Haskell by utilizing its module system and the Yesod Web Framework. Detailled information are listed in chapter 7 "Implementation". In this section I will discuss the particular pieces that form Star-Exec-Presenter into a full server-side web application. To better understand the application structure, it could easily be interpreted as a Model-View-Controller (MVC) pattern where the Model stands for the data model, the View is represented by the templates which form the HTML presentation, and the Controller in the form of the several request handlers.

6.1.1 Data Model

The data model can be devided into two groups, the first containing the data types used for the REST-API which I will discuss later, and the second being the persistent data types of the actual data like the job results.

The first group of data types is used as identifiers for the second group, so they are directly related to each other. For instance there is `JobID` type that relates to `Job` type that containes the infos about a job as well as to a list of `JobResult` containing the actual results of the particular job. All these types mostly are unified types for the different entities of StarExec as well as imported entities.

To complete the example from above the type `JobID` has three different appearences, first being `StarExecJobID` representing a job on StarExec, the second being `LriJobID` representing an imported job from the 2007 Termination Competition and the third being `UibkJobID` for the imported jobs from Innsbruck. Additionally, a `Job` also has three different appearances (`StarExecJob`, `LriJob`, `UibkJob`) as well as `JobResult` (`StarExecResult`, `LriResult`, `UibkResult`).

The reason to have three different kinds of appearances for each type is because of having the data from StarExec in combination with the imported data from previous competitions. This old data differs in its form from that of the 2014 Termination Competition, so to consider these differences in a safe way I decided to isolate them persistent-wise. So, for each kind of data there are three database tables used to store that data.

Code-wise, the data model can be found within the `Presenter.Model` module, which combines all the groups of data types mentioned above as well as the persistent data types, which are defined in the `config/models` file.

6.1.2 Templates

The templates are essentially the view of Star-Exec-Presenter, they are filled with the requested information in a proper way to be displayed via the web browser. HTML[1] is very well suited for this purpose. Other types of data formats like JSON[2], CSV[3] or XML[4] can be considered as well but aren't a subject of matter, although they are used in the communication with the StarExec service.

[1] Hypertext Markup Language, a markup language designed to be interpreted by web browsers
[2] Javascript Object Notation, a compact subset of javascript designed for data exchange
[3] Comma-Separated Values, a data format to represent tables or list
[4] Extensible Markup Language, a data format for hierarchically structured data like trees

All templates can be found in the `templates` folder, some are located within the `Presenter.Utils` module.

6.1.3 Request Handlers

The controller part of Star-Exec-Presenter is represented by the several request handlers. Each handler responds to a certain URL[5] requested by a client application. For instance there are handlers that only return the information of a particular entity like a solver or a job, there are also handler that initiate specific requests to StarExec or display the results of a competition. The request handlers are generally managing the logic behind Star-Exec-Presenter.

All handlers are implemented within the `Handler` module.

6.2 Caching

Fast responses, especially for handlers that can return a huge bunch of data, is a main requirement for Star-Exec-Presenter, so there has to be an effective solution. We came up with the simple idea of caching all data in the database. For instance, when the user requests the results of one or more certain jobs, the application first looks in the database for the respective data and returns it. If there isn't found any related data it will be fetched from StarExec. But this fetching can affect a fast response, so it will be separated from the actual response by running in the background.

This is achieved by starting a new thread within the Star-Exec-Presenter application. The following figure shows the process from a request to a response:

This database based caching mechanism has its downside when it comes the the competition results, as they are calculated based on the results of the related jobs. And as the data of these jobs is stored in the database, there is no need for the results to be stored either. So the results have to be cached in another way which is within the application's memory. For this purpose Star-Exec-Presenter uses STM[6].

The reason to cache the competition results is because it takes too long to fetch the related job results from the database and do the calculations, as a usual competition

[5] Uniform Resource Locator

[6] Software Transactional Memory, a mechanism for concurrent programming with shared resources

has a huge amount of data to process. For instance the 2014 Termination Competition has three meta categories with a total of 19 jobs that produced an overall amount of 37.880 job-pairs. That doesn't seem to be that much but one requirement was to have short response times. So we decided to offload the computation to a separate worker thread. The results are saved within the application's memory and are accessed via STM. The calculating worker thread is the only instance which writes in this cache, all request handlers can only read from it.

STM helps in this case because it can prevent deadlocks from multiple simultaneous access to the shared memory making the whole application more reliable. To achieve this goal every access to the shared memory (our cache of competition results) is being managed by transactions, much like transactions of a database. If two or more transactions try to access a resource simultaneously, all but one will be aborted. A mechanism to retry a rolled back transaction ensures that they will be finished anyhow. To achieve this behavior each transaction must be very small. The actual terminology is *atomic*. It is wise to have very small transactions because STM is not *fair* as it favors small and fast transactions over large and slow ones. STM doesn't work in FIFO[7] order. (Marlow 2013)

Star-Exec-Presenter itself only does the reading and writing process within STM. The calculation of the competition results is done outside of STM. Only in this way we can ensure that the transactions are very small – or *atomic*.

6.3 REST interface

Star-Exec-Presenter is a server-side web application which receives requests and responses with an appropriate HTML presentation. These requests have to point to a valid resource, a URL. A list of important resource-URLs of the application including their accepted HTTP methods follows:

```
1 /jobs/#JobID                    GET
2 /solvers/#SolverID              GET
3 /benchmarks/#BenchmarkID        GET
4 /pairs/#JobPairID               GET
5 /post_procs/#PostProcID         GET
6 /results/#Query/*JobIds         GET
7 /proofs/#Text                   GET
```

[7] First In First Out

```
/registered                          GET
/competitions/#CompetitionInfoId     GET
/control                             GET POST
/import                              GET POST
```

Of course, an explanation of these routes, as they are called, is helpful. The first five routes are self-explaining. They take a specific identifier and return the infos of the related entity, wether it's a job, solver, benchmark, job-pair or post-processor. All these routes have a counterpart which lists all related entities. These routes are accessed without any parameter at the end.

The Results-Route takes a list of Job-Identifiers and returns all of their results. A special part of this route is its first dynamic path piece, the query parameter which defines a special case for this route. If there is given a specific query then the job results will be filtered according to it. More on this topic can be read in the section "Querying".

The Proof-Route is directly related the the Job-Pair-Route. If a job-pair contains a proof, this route is meant to display it. The Registered-Route lists all of the participant of the 2014 Termination Competition including links to their configuration on StarExec. The Competition-Route displays the results for a specific competition. This route has a counterpart, too, which lists all (publicly available) competitions.

The Control-Route is an interface to start a competition wether it's a full or a small one. Requested with the default HTTP method GET this route only displays its interface form. If this form is submitted the POST method is requested and a competition will be started. The Import-Route is used to import old data from previous competitions. It has a small form with a select field and a file input to upload a file. It takes a zip file with the content which has to be defined by the select field.

There are other routes that give further information as well as some routes that are older versions of the aforementioned ones. They are still accessible for legacy purposes and redirect to the new ones.

As Star-Exec-Presenter is meant to be a simple interface for StarExec and the Termination Competition in the first place, these routes are quite simple as well. They only accept GET or POST requests and try to work according to principles of REST. REST is an acronym for Representational State Transfer and was first mentioned by Roy Fielding in his dissertation "Architectural Styles and the Design of Network-based Software Architectures". REST respects the stateless nature of HTTP and

defines that every resource has its unique unified resource identifier (URI). (Fielding 2000)

HTTP methods like GET, POST, PUT or DELETE are used to get, create, change or remove the specific resource. The GET method is nullipotent, which means that a GET request never has any side effects and never changes anything respectively. POST, PUT and DELETE requests are idempotent, resulting in the same outcome no matter how much they are called. The URI, e.g. `/clear_the_database` itself never causes a change to the data unless it is requested with one of the mentioned HTTP methods. Apart from that, this is just an example which isn't recommendable for a real world REST API.

REST utilizes the available web technology for this purpose and isn't based on any standard like SOAP[8]. In contrast to SOAP REST is more like an architectural strategy or design pattern to develop a web application, whereas SOAP is a protocol. REST relies on the hypertext transfer protocol (HTTP) to mediate the request and simplifies the development process with it. To explain this principle a little further I want to emphasize the aforementioned MVC design pattern. In this context a REST interface can be considered the controller of the application which responses to the request with a suitable presentation.

A suitable presentation is HTML as well as XML or JSON. Star-Exec-Presenter currently only sends HTML. But this behavior could be altered by using a special header in the HTTP request. So, to get the data as XML or JSON, the Accept-Header could be set to the values `Accept: text/xml` or `Accept: text/json`. The web application then can evaluate this header to respond either with an XML or with a JSON presentation. (Richardson 2007)

It is widely accepted not to use verbs as parts of the URL rather than plural nouns. For instance the route `/get_job_results/#JobID` could be better written as `/job_results/#JobID` or simply `/results/#JobID`. Also, it is recommended to have a generalized route, e.g. `/jobs`, which lists all resources, whereas a parameter added to the URL results in returning only the specified resource. In the case of Star-Exec-Presenter especially the Results-Route is a special case takes an arbitrary amount of `JobID`s.

6.4 Querying

[8]Simple Object Access Protocol, a web protocol for structured information

7

Implementation

The Star-Exec-Presenter is a web application written in Haskell and based upon the Yesod Web Framework. In this chapter I will talk about the actual implementation of the tool as well as about the benefits of using especially Haskell and Yesod.

7.1 Haskell

"Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on non-strict functional languages." (Peyton Jones 2003, Marlow (2010))

This quote stems from Simon Peyton Jones and his paper "Haskell 98 language and libraries: the revised report" as well as from the paper "Haskell 2010 Language Report" by Simon Marlow. It does sum up the important features of the Haskell programming language. In addition I want to add that Haskell has type inference and its type system is very expressive. In the following I try to explain the listed terms.

Haskell is a general purpose programming language which means it can be used to

develop a web-server, a desktop application as well as a simple commandline tool. It is suited for a wide range of application domains. Haskell is a purely functional programming language, that is, it follows the functional programming paradigm and every pure function has no side effect. Functional programming relies on a mathematical approach where every calculation is defined by expressions. Everything is an expression, wether it's a function or a value. Purely means, that a function only works with its input arguments and returns the same result no matter how often it is called with the same arguments. Every expression is immutable so once they are defined they cannot be changed anymore. Haskell code therefore is easily testable and even proofable. In contrast, imperative programming languages allow mutability and functions are subroutines that can have side effects. A side effect for instance can be a simple tracing that doesn't effect the function's result or it can be the change of a global state.

```haskell
-- example of a pure function
mul a b = a * b

-- example of an impure function
mulM a b = do
    putStrLn ("multiplying " ++ (show a) ++ " with " ++ (show b))
    return (a * b)
```

It is important to note that, unlike programming languages like Java or C, Haskell is indentation based. So, there are no blocks surrounded by curly brackets, but every whitespace at the beginning of each line has a meaning to the program.

7.1.1 Higher-Order Functions

Haskell has higher-order function and emphasizes their usage. Higher-order function are functions that require functions as input parameters or return a function. Therefore functions are data that can be passed round. Thanks to currying all functions are higher-order functions and thereby expressions. Currying means that a function always returns a new function if it isn't called with the full list of arguments.

```haskell
-- example for a higher-order function
map :: (a -> b) -> [a] -> [b]
map _ []     = []               -- special case for an empty list
map f (x:xs) = f x : map f xs   -- f applied to the list's head
                                -- recursive call of map with
```

```
6                                   -- the list's rest
```

7.1.2 Pattern-Matching

The example from above also shows pattern-matching, a rich programming technique to identify certain cases for a given expression. The `map` function from above has two implementations, one for the case of the empty list which also is the base case for the recursion, and the main implementation that declares the algorithm of the function. Pattern matching helps simplifying code by reducing the usage of if-statements and alike.

7.1.3 List Comprehensions

To stay with the `map` function, list comprehensions are another way to define this method as well as lists in general. They are similar to set comprehensions in mathematics. For example the list comprehension `[ 2*x | x <- [1..10] ]` is equivalent to the set comprehension $\{2x | x \in \mathbb{N}, x \leq 10\}$. (Lipova a 2012) Both expressions result in a list of even natural numbers from two to 20. Referecing the `map` function from above, an alternative implementation follows:

```
1 map f xs = [ f x | x <- xs ]
```

Another example of list comprehensions is the solution to the first problem of Project Euler (https://projecteuler.net/problem=1). The exercise is to find the sum of all multiples of three or five below 1000:

```
1 sum [x | x <- [3..999], x `mod` 3 == 0 || x `mod` 5 == 0]
```

7.1.4 Non-Strict Semantics, Lazy Evaluation

Another essential feature of Haskell is the non-strict semantics. Non-strict in terms of Haskell means that every expression is evaluated by need. So, only declaring an expression doesn't invoke its evaluation until it's needed, e.g. for output. For instance the fibonacci sequence can be seen as an infinite list. This sequence can also be implemented via a list comprehension:

```
1 let fibs = 0 : 1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

Here, we have the declaration of an infinite list, but which will be evaluated only up until its seventh position. These non-strict semantics in Haskell are implemented as *lazy evaluation* which leeds to the expression from above also being evaluated by need. So until it is really needed, `fibs` is only a thunked[1] expression. But this strategy can lead to high memory usage especially for complex algorithms, so in some cases it is advised to use strict evaluation which can be used with the `seq` function in Haskell's `Prelude` module. Other modules with strict evaluations for example are `Data.Map.Strict` or `Data.List` the latter having a strict implementation of the `Prelude`'s `foldl` function. Especially this function can lead to a stack overflow with a large list.[2]

7.1.5 Type System, Type Inference

Haskell has a rich type system which can automatically infer the type of an expression. For example the fibonacci sequence from above has the type `fibs :: Num b => [b]` which was inferred bei the `(+)` operator's type (`(+) :: Num a => a -> a -> a`). Not all types can be inferred and the compiler outputs a compiler error, so a type signature should be added. In most cases these signatures are optional but also can help improving the readability of the code. These signatures doesn't contain any special type information because they are polymorphic, that is, they can be used with any type that meets the functions requirements, may it be `Int` or `Double`. Polymorphic types are denoted by small alphanumeric characters like `a`, `b` or `t0`. They can be restricted by a type-class which is a set of functions. The `(+)` operator is a function of the `Num` type-class which is a set of functions to operate on numerical values. Type-classes are similar to interfaces of imperative programming languages like Java.

7.1.6 Algebraic Datatypes

With Haskell a developer can declare custom algebraic datatypes. What that means is that a type can be defined by the algebraic operations *sum* and *product*, where a *sum* is a group of alternative data constructors and *product* is the combination of them. Below are some examples:

```
1 -- sum of polymorphic types
```

[1] a value that is yet to be evaluated
[2] see https://www.haskell.org/haskellwiki/Foldr_Foldl_Foldl%27

```haskell
data Either a b = Left a | Right b

-- product of polymorphic types
data Pair a b = Pair a b

-- a record datatype
data Record a b =
  Record
    { memberA :: a
    , memberB :: b
    }
```

The `Either` datatype is a special type that is used for computations that can result in an error, where the `Left` constructor contains the error and the `Right` constructor holds the successful result. This leads to Haskell's expressiveness which is one of the benefits of its type system. The `Either` datatype explains that a function can return an error, so a developer has to consider the returned value. Java in contrast is more verbose by forcing to add a `throws` statement to the method definition. A developer has to use a `try ... catch` block to manage an error. Another example for the expressive type system of Haskell is the `Maybe` datatype whose signature follows:

```haskell
data Maybe a = Nothing | Just a
```

This datatype is used for computations that can have no result or that require optional arguments. The `Nothing` constructor means that there is no value, whereas the `Just` constructor wraps a result or an optional argument. The `Maybe` is usefull because it prevents `undefined` or `null` values via the type system. In Java, for example, careless developed programs can pass the compiler but also result in a `NullPointerException` whose cause can be difficult to resolve. Of course, that doesn't mean that a Haskell program doesn't have errors, but the type system and the compiler helps to reduce them or event prevent them all.

7.1.7 Modules

Haskell has a distinct module system, where each module contains a well-matched set of functions, types and type-classes. The Haskell code base is separated into a large number of modules, each one serving a certain kind of purpose. Each Haskell program has a main module which contains the main function as well as a set of other

modules. It is important to know that, without any additional language extensions, a bunch of modules cannot be defined circularly, that is module A cannot import module B it this already imports module A.

7.1.8 Input/Output, Monads

All these features and benefits of Haskell are of no use if it can't communicate with the real world, that is doing input/output. So, Haskell has an appropriate I/O system which is also monadic[3]. That means that every function that operates within the IO-Monad has a special type signature. Below are some examples:

```
putChar     :: Char -> IO ()
putStrLn    :: String -> IO ()
getLine     :: IO String
readFile    :: FilePath -> IO String
writeFile   :: FilePath -> String -> IO ()
```

All functions return a value within the context of the IO-Monad. To access the content it must be bound to a function that processes it. For example, the `putStrLn` function can be implemented via the `putChar` function and a monadic composition ((>>)):

```
putStrLn' s = mapM_ putChar s >> putChar '\n'
```

This implementation uses the monadic version of the `map` function to put each `Char` of the given `String` to the output followed by a newline. To simplify the work with the IO-Monad or Monads in general Haskell has the do-notation which I want to explain with the following example:

```
func_do f g h = do
    a <- f              -- binds the wrapped value of f to a
    b <- g              -- binds the wrapped value of g to b
    c <- h              -- binds the wrapped value of h to c
    return (a, b, c)    -- wraps the result in the default context

func_bind f g h =
    f >>= \a ->
        g >>= \b ->
```

[3] A Monad is an abstract datatype, that wraps a (primitive) value in a context and binds it to a function operating within that context

```
10              h >>= \c ->
11                  return (a, b, c)
```

Both functions essentially do the same. They take three monadic values (e.g. `Maybe a`) and bind the content of these values to `a`, `b` and `c`. The first function uses do-notation, the second one uses the bind-Operator (`(>>=)`). The do-notation is a sugared way of using the bind-Operator, that is the compiler transforms the code of `func_do` to the code of `func_bind`. It is the same with all functions of the IO-Monad. But there is one important note: The IO-Monad is impure. What does this mean? All functions in Haskell are pure unless they use the IO-Monad. Because this Monad communicates with the real world, Haskell has no control of this communication. That means, that every call of an IO-function the result is not garanteed to be the same. A File could have changed since the last time it was read. Another good example is the operation system's random number generator. Each time, it produces another value, so this process can never be pure.

(O'Sullivan, Goerzen, and Stewart 2010)

7.2 Yesod Web Framework

The Yesod Web Framework is a set of tools and libraries build with Haskell. It aims to improve the development of server-side, RESTful web applications by providing type-safety, conciseness and performance. It is devided into several components that together form the final web application. Yesod consists of an object-relational mapping for the communication with a database, several domain-specific languages (DSLs) for routes, templates and the database types, it has session- and forms-handling as well as the support for authentication, authorization or internationalization. All DSLs are implemented with Template Haskell, an extension to the GHC[4] to enable metaprogramming. Template Haskell code will be transformed to actual Haskell while compiling. (Sheard and Jones 2002)

The basis of Yesod is the Warp server which is a fast web server build in Haskell. It is the main implementation of the Web Application Interface which in turn is generalized interface for building web server. It generally considers the nature of the web infrastructure, where the communication essentially works like a function: a client sends a request und gets a response in return. This communication can be illustrated with the following type signature:

[4]Glaskow Haskell Compiler, the most used compiler infrastructure for Haskell

```
askServer :: Request -> Response
```

WAI aims to be an efficient interface by relying an lazy IO, which leads to a small memory footprint. An example of a small WAI implementation follows below:

```
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.HTTP.Types (status200)
import Network.Wai.Handler.Warp (run)

application _ respond = respond $
  responseLBS status200 [("Content-Type", "text/plain")] "Hello
      World"

main = run 3000 application
```

This program takes a request and responses with the text "Hello World", regardless of what the request may contain. A more complex example considers the requested path (route) and responds according to that:

```
{-# LANGUAGE OverloadedStrings #-}

import Network.Wai
import Network.Wai.Handler.Warp
import Network.HTTP.Types (status200)
import Blaze.ByteString.Builder (copyByteString)
import qualified Data.ByteString.UTF8 as BU
import Data.Monoid

main = do
  putStrLn $ "Listening on port 3000"
  run 3000 app

app req respond = do
  let res = case pathInfo req of
        ["yay"] -> yay
        ["foo","bar"] -> foobar
        x -> index x
  respond res
```

```
20
21 yay = responseBuilder status200
22   [ ("Content-Type", "text/plain") ]
23   $ mconcat $ map copyByteString [ "yay" ]
24
25 index x = responseBuilder status200
26   [("Content-Type", "text/html")] $ mconcat $ map copyByteString
27   [ "<p>Hello from ", BU.fromString $ show x, "!</p>"
28   , "<p><a href='/yay'>yay</a></p>\n" ]
29
30 foobar = responseBuilder status200
31   [("Content-Type", "text/html")] $ mconcat $ map copyByteString
32   [ "<p>Hello from foobar! ;)</p>" ]
```

7.2.1 Routes

Yesod has a more complex mechanism for handling requests build atop of WAI and Warp. The routes are defined via a special DSL which concentrates on the path and method of each request. A path is devided into one or more path pieces each one being static or dynamic, where dynamic parts are representated by a distinct type. There can be can be a single dynamic path piece as well as any number. Each path is followed by the handler resource and the supported method which can be a standard HTTP method or even a custom one. The handler resource is the connection to the Haskell code. Template Haskell generates a handler function for each route and method as well as special datatype representing the route. This datatype enables typesafe URLs. In the section "REST interface" in chapter six I showed an example of the routes DSL. Below is an explaining example:

```
1 -- the home route
2 -- generates the handler "getHome"
3 -- "HomeR" is the datatype for this route
4 /               HomeR       GET
5
6 -- the list of users
7 -- generates the handler "getListUsers" and "postListUsers"
8 -- "ListUsersR" is the datatype for this route
9 /users          ListUsersR  GET POST
```

```
10
11 -- generates the handler "getUser" and "postUser"
12 -- "UserR UserId" is the datatype for this route
13 /users/#UserId  UserR       GET POST
14
15 -- generates the handler "getList"
16 -- "ListR [Text]" is the datatype for this route
17 -- "Texts" is a type synomym for "[Text]"
18 /list/*Texts    ListR       GET
19
20 -- special routes for static files and authentication
21 /static StaticR Static getStatic
22 /auth   AuthR   Auth   getAuth
```

7.2.2 Templates

Yesod uses templates for HTML, JavaScript and CSS[5] as type-safe expressions which are inserted into the application at compile time. Again, Template Haskell takes care of them. Templates can are inserted into the handler functions via Quasi-Quoters, some of which take the template code while others need to get the filename of the template. So, Yesod is very flexible.

In Yesod all template DSLs are named after a Shakespear charactor. For HTML, there is the Hamlet markup language:

```
1 <div>
2   <h1>#{myTitle}
3   <p>
4     <a href="@{MyCustomRouteR}">my custom route</a>
5   $maybe val <- maybeVal
6     <p>val
7   $nothing
8     <p>There is Nothing to show here
9   $if null someList
10    <ul>
11      $forall item <- someList
12        <li>#{show item}
```

[5]Cascading Stylesheet

```
13   $else
14     <p>The List is empty
15 ^{footerWidget}
```

As you can see, the syntax is very similar to real HTML. But, it is important to note, that Hamlet is mostly indentation based. *Mostly* means, that indentation is used for block elements like `div`, `p` or headlines, so these elements doesn't have a closing tag in Hamlet. Actually they would cause a compiletime error. Inline elements such as anchor tags (`<a href=""></a>`) must have a closing tag. The integration with actual Haskell code has several incarnations. The first example is the interpolation of the expression `myTitle` which is bracketed by `#{...}`. It will be inserted into the the HTML code with the `toHtml` function of the `ToHtml` type-class. So every expression must be an instance of this type-class to be interpolated. The next feature of Hamlet is typesafe URLs which incorporates with the routes definition. The resource datatypes from the routes DSL can be used here. So an expression bracketed by `@{...}` generates a typesafe URL within a `href` attribute. This is possible for all attributes containing a path to a resource.

Another feature is the special treatment of `Maybe` values as well as lists. Here, special statements intialize Haskell code which handles the values. So a `Maybe` value will be unwrapped (`$maybe`) or an optional information can be shown if there is `Nothing` (`$nothing`). The same applies to lists where the statement `$forall` indicates a loop to process all items of a list. Additionally, there are logical statements like `$if/$else` as well as `$case/$of`, the latter also supporting pattern matching. And to shorten inconveniently long expressions a new one can be declared via `$with shortExp <- longExp`.

The shakespearean templates are round out with the DLSs Julius for JavaScript and Lucius as well as Cassius for CSS. Julius is basically JavaScript with the features of interpolation. Lucius and Cassius are both an optimized subset of CSS equivalent to the likes of SASS[6] or Less[7]. They feature nested blocks, interpolated expressions, variable declaration as well as mixins. Mixins are reusable code blocks, e.g. for dealing with vendor prefixes. Whereas Lucius is more like original CSS with curly brackets enclosing the CSS statements, Cassius is intendation based. Below is an code example for Lucius taken from Star-Exec-Presenter:

```
1 @colorYes: #80FFB0;
2 ...
```

[6] a dynamic stylesheet language for CSS (http://sass-lang.com/)
[7] a dynamic stylesheet language for CSS (http://lesscss.org/)

```
3
4 .table > tfoot > tr {
5     > td.solver-yes,
6     > th.solver-yes {
7         background-color: #{colorYes};
8     }
9     ...
10 }
```

7.2.3 Persistence

For the communication with and the integration of a database, Yesod relies on the persistent framework. This framework enables a generalized way of working with the database regardless of its kind, may it be SQL or NoSQL. Even for persistent datatypes there is a DSL which simplifies their declaration as there a many special types incorporating. Below is an example:

```
1 User
2     ident Text
3     password Text Maybe
4     name Text
5     gender Gender
6     UniqueUser ident
7     deriving Show
```

This is a simple example of representation of a user in the database. Just like most DSLs, this one is indentation based, too. It declares the type `User` with four member and a uniqueness constraint. Each member is defined by its name followed by its type. Unlike Haskell where an optional type is defined by `Maybe a`, the persistent DSL requires `Maybe` to be at the end of the declaration. A unique constraint is defined with the prefix `Unique` to the type's name as well as the names of one or more members of the type. A full description of the persistent DSL syntax can be found in the wiki of the `persistent`-repository on GitHub[8].

From this definition several types are generated. First and foremost a record datatype will be defined representing the actual type. This datatype is made instance of several type-classes to make them compatible with the persistent interface. Additionally a

[8]https://github.com/yesodweb/persistent/wiki/Persistent-entity-syntax

data constructor is available to represent a unique instance of the data in the database. There are also several datatype for filtering the data table by specific constraints. Custom type, such as `Gender` must be an instance of the type-class `PersistField` which takes care of serialization and deserialization of the custom datatype. An example outcome follows:

```
data User = User { userIdent :: Text
                 , userPassword :: Maybe Text
                 , userName :: Text
                 , userGender :: Gender }
           deriving (Show)

-- datatype representing the primary key of the table
-- will be serialized to the actual database type value
data UserId = KeyBackend SqlBackend User
```

To actually communicate with the database the type-classes `PersistStore`, `PersistUnique` and `PersistQuery` are used. All three offer usefull functions to query and store all related data in the database. The following listing shows some possible usages:

```
import qualified Data.Text as T

communicateWithDB = runDB $ do
    let ident = T.pack "haskell@example.org"
        name = T.pack "Haskell Curry"
        gender = Male
        user = User ident Nothing name gender
    -- inserting the user entity
    -- but could cause an error
    -- if the user is already in the database
    userId <- insert user
    -- because of uniqueness constrain
    -- insertUnique is better suited
    mUserId <- insertUnique user
    case mUserId of
        Just userId -> print userId
        Nothing     -> error "user already in db"
    -- setting the password of the user
```

```
19    let passwd = Just $ T.pack "someSecretPassPhrase"
20    update userId [ UserPassword =. passwd ]
21    -- get unique user entity
22    mUser <- getBy $ UniqueUser ident
23    case mUser of
24        Just user -> print user
25        Nothing   -> error "No user found"
26    -- counts all users by a certain filter
27    numberOfMales <- count [ UserGender ==. Male ]
28    -- fetches 10 users by a certain filter
29    femaleUsers <- selectList
30                   [ UserGender ==. Female ]
31                   [ Asc UserName
32                   , LimitTo 10
33                   , OffsetBy 0 ]
34    -- deleting a complete table
35    deleteWhere ([] :: [Filter User])
```

The `runDB` function wraps all actions within a transaction. Actually, each call of `insert` or `selectList` can be called on its own with `runDB`. But to make the code more efficient, it is useful to put all steps into one call to `runDB`. Another advantage is that an error rolls back the full transaction.

(Snoyman 2012)

8 Evaluation

8.1 Future considerations

- caching could be automated with a permanent background-worker which traverses the database's content after a given time period
- ajax-based implementation of the web-interface
- generalized export of all data

9

Summary

…

Glossary

References

Fielding, Roy Thomas. 2000. *Architectural Styles and the Design of Network-Based Software Architectures*. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Lipova□ a, Miran. 2012. *Learn You a Haskell for Great Good!: a Beginner's Guide*. San Francisco, CA: No Starch Press.

Marché, Claude, and Hans Zantema. 2007. "The Termination Competition." In *In Proc. RTA '07, LNCS 4533*, 303–13.

Marlow, Simon. 2010. "Haskell 2010 Language Report." https://www.haskell.org/definition/haskell2010.pdf.

———. 2013. *Parallel and Concurrent Programming in Haskell*. First edition. Beijing: O'Reilly.

O'Sullivan, Bryan, John Goerzen, and Donald Bruce Stewart. 2010. *Real World Haskell*. Auflage: 1. Sebastopol, CA: O'Reilly.

Peyton Jones, Simon L., ed. 2003. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge, U.K. ; New York: Cambridge University Press.

Richardson, Leonard. 2007. *RESTful Web Services*. Farnham: O'Reilly.

Sheard, Tim, and Simon Peyton Jones. 2002. "Template Meta-Programming for Haskell." In *In Proceedings of the ACM SIGPLAN Workshop on Haskell*, 1–16. ACM.

Snoyman, Michael. 2012. *Developing Web Applications with Haskell and Yesod*. 1st ed. Sebastopol, CA: O'Reilly.

StarExec. 2013. "About StarExec." https://www.starexec.org/starexec/public/about.jsp.

Stump, Aaron, Geoff Sutcliffe, and Cesare Tinelli. 2014. "StarExec: A Cross-Community Infrastructure for Logic Solving." In *Automated Reasoning*, edited by

Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, 8562:367–73. Lecture Notes in Computer Science. Springer International Publishing. http://dx.doi.org/10.1007/978-3-319-08587-6_28.

Waldmann, Johannes. 2014. "Termtools Termination Competition Data." http://lists.lri.fr/pipermail/termtools/2014-July/000979.html.

Zantema, Hans. 2000. *Termination of Term Rewriting*.