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.

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)

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 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[2] with the utalization of the Yesod Web Framework[3].

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.

With Star-Exec-Presenter we wanted to extend the ability of the StarExec service with filtering of the results and comparing them with older ones from previous competitions. Contributers of solvers should be able to track their progress of development of their respective solver through time.

[1] Representational state transfer, a paradigm to implement a server-client communication
[2] a functional, non-strict, declarative programming language (http://www.haskell.org/)
[3] 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.

Termination of Term Rewriting

(Zantema 2000)

Termination Community

Termination Competition

(Marché and Zantema 2007)

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.

Interfaces

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.

Displaying Results and Outputs

The core 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.

Compare Results

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.

Querriyng

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.

Interpret Outputs as possible XML-Proofs

Some solver explicitly log an 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 should be used to display the proof in the browser.

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.

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.

5

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.

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.

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.

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.

6

Star-Exec-Presenter

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.

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, that the application uses the 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.

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 linked 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.

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.

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.

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 resprective 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 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

[5] Uniform Resource Locator

[6] Software Transactional Memory, a mechanism for concurrent programming to prevent deadlocks

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 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, 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. Everything else, e.g. 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*.

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:

`/job/#JobID`	`GET`
`/solver/#SolverID`	`GET`
`/benchmark/#BenchmarkID`	`GET`
`/pair/#JobPairID`	`GET`
`/post_proc/#PostProcID`	`GET`
`/results/#Query/*JobIds`	`GET`
`/proof/#Text`	`GET`
`/registered`	`GET`
`/competition/#CompetitionInfoId`	`GET`

[7] First In First Out

```
/control                        GET POST
/import                         GET POST
```

Of course, an explanation of these routes, they are called, is helpfull. 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. 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.

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.

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.

As Star-Exec-Presenter is meant as 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 REST.

(Richardson 2007)

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.

Haskell

(O'Sullivan, Goerzen, and Stewart 2010) (Peyton Jones 2003) (Marlow 2010)

Yesod Web Framework

(Snoyman 2012)

Routes

Templates

Persistence

8

Evaluation

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

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. 2003. "Haskell 98 Language and Libraries The Revised Report." https://www.haskell.org/definition/haskell98-report.pdf.

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

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.

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

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