Fachbereich 4: Informatik

# Haskell Programming Technologies

## Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von
Thomas Schmorleiz

| | |
|---|---|
| Erstgutachter: | Ralf Lämmel<br>Institut für Informatik |
| Zweitgutachter: | Andrei Varanovich<br>Institut für Informatik |

Koblenz, im November 2011

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☐ | ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ☐ | ☐ |

.....

(Ort, Datum) (Unterschrift)

# Abstract

In this thesis we exercise a wide variety of libraries, frameworks and other technologies that are available for the Haskell programming language. We show various applications of Haskell in real-world scenarios and contribute implementations and taxonomy entities to the 101companies system [1]. That is, we cover a broad range of the 101companies feature model [2] and define related terms and technologies. The implementations illustrate how different language concepts of Haskell, such as a very strong typing system, polymorphism, higher-order functions and monads, can be effectively used in the development of information systems. In this context we demonstrate both advantages and limitations of different Haskell technologies.

# Zusammenfassung

In der vorliegenden Arbeit untersuche ich eine breite Spanne von Libraries, Frameworks und anderer Technologien für die Programmiersprache Haskell. Ich demonstriere verschiedene praktische Anwendungen von Haskell und trage durch Implementationen und Taxonomieeinheiten zum 101companies System bei [1]. Dies tue ich, indem ich einen großen Teil des 101companies feature models [2] abdecke und damit verknüpfte Defintionen von Begriffen und Technologien in dieses System einpflege. Die Implementationen zeigen, wie verschiedene Sprachkonzepte von Haskell wie ein sehr starkes Typensystem, Polymorphismus, Funktionen höherer Ordnungen und Monaden effektiv bei der Entwicklung von Informationssystemen benutzt werden können. In diesem Zusammenhang zeige ich sowohl Vorteile als auch Einschränkungen der verschiedenen Technologien auf.

# Acknowledgements

I gratefully acknowledge collaboration with Ralf Lämmel (Software Languages Team, Koblenz) and Andrei Varanovich (Software Languages Team, Koblenz) on the underlying 101companies software system.

# Contents

# Chapter 1

# Introduction

## 1.1 Why Haskell?

Over the past few years the attention to the Haskell programming language has strongly increased. One fundamental reason for this is the wide variety of libraries, frameworks and other technologies that are available for Haskell covering domains from database programming to GUI development. Another reason for the recent attention is the continuous development of concurrency and data parallelism support. Technologies like the concurrency concept of MVars [3] and the GHC extension Data Parallel Haskell [4] provide high performance tools for concurrent and parallel programming at a high level of abstraction. This thesis' implementations *haskellConcurrent* and *dph* demonstrate MVars and DPH.

New technologies that are available for Haskell are published on Hackage [5], a database for Haskell packages and a centralized documentation infrastructure. In combination with the package management system Cabal [6], this provides an easy way to download and install new Haskell technologies.

In this thesis we will demonstrate some of Haskell's language features [7]: Haskell programs are easy to write and read because of an expressive elegant syntax [8] and a powerful type inference system [9]. We illustrate this in various implementations. Haskell functions are pure, that is, they have no side effects. Any I/O actions, like file reading and writing, database transactions and networks connections, are encapsulated inside a special monad [10]. This approach strictly separates the I/O actions from the pure computations of the program. We will benefit from this feature in several real-world application like web programming (covered by the implementations *haskellCGI* and *happstack*) and XML processing (covered by *hxt* and *hxtPickler*). Haskell programs are very modular because one can divide the tasks of a program into small functions. This helps us to provide reusable code and to avoid redundancy [7]. Haskell is a lazy programming language, which means that expressions are only evaluated when they are actually used [11]. Because of lazy evaluation we can realize streaming-based processing of data sets in databases (demonstrated in *hdbc*) [12]. We are making use of monads in Haskell to compose parsers in *haskellParser* and to realize logging in *haskellLogger*.

## 1.2 Coverage of the feature model

In this thesis we exercise various Haskell technologies covering most features of the 101companies feature model [2]:

Table 1.1: Coverage of the feature model

| | basics | | | capabilities | | | | | | | | | | | | | extras | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Company | Cut | Total | Access control | Concurrency | Distribution | Fault tolerance | Interaction | Logging | Mapping | Parallelism | Parsing | Persistence | Serialization | Validation | Visualization | Depth | Mentoring | Precedence |
| haskellLogger | • | • | • | | | | | | • | | | | | | | | | | |
| haskellParser | • | • | • | | | | | | | | | • | | • | | | | • | |
| haskellConcurrent | • | • | • | | • | | | | | | | | | | | | | | |
| dph | • | • | • | | | | | | | | • | | | | | | | | |
| hdbc | • | • | • | | | | | | | • | | | • | | | | | | |
| haskellDB | • | • | • | | | | | | | • | | | • | | | | | | |
| hxt | • | • | • | | | | | | | • | | | | • | | | | | |
| hxtPickler | • | • | • | | | | | | | • | | | | • | | | | | |
| wxHaskell | • | • | • | | | | | • | | | | | | | | | | | |
| haskellCGI | • | • | • | | | • | | • | | | | | | | | | | | |
| happstack | • | • | • | | | • | | • | | | | | | | • | | | | |
| **Coverage** | **11** | **11** | **11** | **0** | **1** | **2** | **0** | **3** | **1** | **4** | **1** | **1** | **2** | **3** | **1** | **0** | **0** | **1** | **0** |

- All implementations cover the basic features of modeling a company, totaling all salaries in a company and cutting all salaries.
- Logging the process of totaling and cutting salaries by means of the Writer monad is performed by *haskellLogger*.
- The *haskellParser* implementation covers the features of parsing concrete company syntax and the extra feature of mentoring.
- *haskellConcurrent* implements concurrent salary processing by using MVars.
- We exercise parallelism in the Data Parallel Haskell implementation *dph* .
- Persisting company data in a database is realized in the *hdbc* and *haskellDB* implementations. In the former we execute SQL-statements represented as strings in Haskell. In the latter we make use of a combinator library to express queries as Haskell functions.
- *hxt* and *hxtPickler* serialize companies as XML data and process such data.
- *hxtPickler* also covers mapping XML data to values of algebraic datatypes for companies. *hxt*, *haskellDB* and *hdbc* map query results to values in Haskell.
- The feature of interacting with a company by means of an user interface is covered by the GUI-based standalone implementation *wxHaskell* and the web applications *haskellCGI* and *happstack*.
- *haskellCGI* and *happstack* are also providing distribution of company data.
- *happstack* additionally performs validation of user input.

## 1.3 Structure of implementation documentations

In the next chapter we will describe this thesis' set of implementations. These documentations all follow a common structure, which is described in [13]. That is, we start off with an intent describing the implementation at hand in a short phrase. We then list all used languages and technologies. For instance, all implementations list Haskell as a language and mention the Haskell compiler GHC or the Haskell interpreter GHCi as a technology. After that we list all implemented features as described in the last section. Followed by that is a short motivation discussing the contribution of the implementation to the 101companies software corpus. We then illustrate idiomatic code of the implementation. The usage section describes how one can run or deploy the implementation. Additionally some documentations list issues of the current status of the implementation in question. All sections use links to the 101companies wiki [14] and links to this thesis' appendix, which defines terms and technologies.

# Chapter 2

# Implementations

## 2.1 *101implementation* haskellLogger

### 2.1.1 Intent

Logging in Haskell by means of the Writer monad

### 2.1.2 Languages

- Haskell

### 2.1.3 Technologies

- GHCi

### 2.1.4 Features

- Company
- Total
- Cut
- Logging

### 2.1.5 Motivation

We exercise logging in Haskell by making use of the Writer monad. That is, during the process of totaling and cutting companies we log messages regarding intermediate results. In this implementation we choose that logs should be of type `[String]`, yet they could be of any monoid type.

### 2.1.6 Illustration

#### Logging cutting

We provide functionality for cutting all company, department and employee salaries. In the following we will show how cutting all salaries in a given department is realized in the current implementation.

Listing 2.1: Cut.hs

```
1  cutLogDept :: Int -> Department -> Writer Log
       Department
2  cutLogDept n d@(Department name m dus eus) = do
3      tell [replicate n '\t' ++ "Starting cutting "
4                             ++ "department \""
5                             ++ name
6                             ++ "\", old Total = "
7                             ++ (show $ totalDept d)]
8      cutManager <- cutLogEmployee (n + 1) m
9      cutDus <- mapM (cutLogDept (n + 1)) dus
10     cutEus <- mapM (cutLogEmployee (n + 1)) eus
11     let cutD = Department name cutManager cutDus cutEus
12     tell [replicate n '\t' ++ "Done cutting "
13                            ++ "department \""
14                            ++ name
15                            ++ "\", new Total = "
16                            ++ (show $ totalDept cutD)]
17     return cutD
```

In line 3 we log the start of the process of cutting a department by adding a message containing the department's name and the old total salary. We make use of the `tell` function, which is provided by the *Control.Monad.Writer* module, to add messages to the log. To prettyPrint this log we indent all log lines using the given indent size `n`. In line 8 we cut the department manager's salary by passing the manager and an increased indent size to `cutLogEmployee`. To cut all sub departments and employees we make use of the monadic map function **`mapM`** in lines 9 and 10. In the following lines we log that department cutting is finished and what the new total salary is. In line 17 we return the cut `Department` value.

### 2.1.7 Architecture

Total.hs and Cut.hs contain functionality to total and cut salaries while logging the process of doing so. Types.hs holds the log type and a function for prettyprinting logs. The algebraic datatype for companies can be found in Company.hs. Main.hs collects test scenarios for totaling and cutting a sample company provided by SampleCompany.hs.

### 2.1.8 Usage

- Main.hs has to be loaded into GHCi.

- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* for test automation.

## 2.2 *101implementation* haskellParser

This implementation is a joint work with Ralf Lämmel and Martijn van Steenbergen.

### 2.2.1 Intent

Parsing textual syntax with Parsec in Haskell

### 2.2.2 Languages

- Haskell

### 2.2.3 Technologies

- GHCi
- Parsec

### 2.2.4 Features

- Company
- Total
- Cut
- Mentoring
- Serialization
- Parsing

### 2.2.5 Motivation

We make use of Haskell's Parsec parser combinator library to parse concrete textual syntax for companies. We combine smaller parsers, say for salaries, to larger parsers, say for departments and employees, to build a parser for companies. In terms of parsing we exercise sequence, alternative and option. This implementation also demonstrates applicative functors and functor combinators provided by the *Control.Applicative* module.

### 2.2.6 Illustration

#### Parser type

We define a type alias of all parsers that are defined in this implementation:

Listing 2.2: Parser.hs

```
1 type P = Parsec String ()
```

That is, we are dealing with parsers of stream type `String` and state type `()` (no state). The return type of running such a parser is explained further below.

#### Primitive parsers

In order to build the company parser we first need some primitive parsers. For parsing a given `String` value we define:

Listing 2.3: Parser.hs

```
1 pString :: String -> P String
2 pString s = string s <* spaces
```

This parser also consumes trailing spaces. We also need a parser for literals. `pLit` parses a quoted string:

Listing 2.4: Parser.hs

```
1 pLit :: P String
2 pLit = string "\"" *> many (noneOf "\"") <* string "\""
       <* spaces
```

#### Parsing a department

Listing 2.5: Parser.hs

```
1 pDepartment :: P Department
2 pDepartment = Department
3   <$  pString "department" <*> pLit
4   <*  pString "{" <*> pEmployee "manager"
5   <*> many pSubUnit <* pString "}"
```

We make use of the `(<$) :: Functor f ⇒ a → f b → f a` operator in line 3. That is, we pass the department constructor `Department` and a parser for all constructor parameters (for name, manager and for the list of subunits) to receive a parser for departments. In line 3 we parse the keyword for department declaration ”department”.

In the next line we parse the department name, followed by an opening curly bracket. We compose a parser for the department's manager using the employee parser `pEmployee`, which should use "manager" as the keyword. In the last line `many :: f a → f [a]` is used to parse the list of subunits using `pSubUnit` as the parser for each subunits. Finally, we define that we expect a closing curly bracket at the end of a department declaration.

### Running the Parser

Running the company parser is realized by making use of `runP`:

```
runP :: Stream s Identity t => Parsec s u a -> u ->
    SourceName -> s -> Either ParseError a
```

We illustrated above that we use parsers of type `Parsec String ()`, which on the top level parse values of type `Company`. Therefore we can simplify the type signature:

```
Parsec String () Company -> () -> SourceName -> String
    -> Either ParseError a
```

When we choose `"input"` as the source name we can define a function to run the company parser:

Listing 2.6: Parser.hs

```
parseCompany :: String -> Either ParseError Company
parseCompany = runP (spaces *> pCompany <* eof) ()
                "input"
```

We added a parser for possible leading spaces and a parser for the EOF-symbol.

### Executing the Parser

In Main.hs we execute the company parser:

Listing 2.7: Main.hs

```
parsedCompany <- liftM parseCompany $
                 readFile "sample.Company"
```

The variable `parsedCompany` either holds a `ParseError` value or a parsed company. We define a function for printing, which handles both cases:

Listing 2.8: Main.hs

```
eitherPrint :: Show a => Either ParseError Company -> (
    Company -> a) -> IO ()
```

```
2 eitherPrint (Right c) f = print $ f c
3 eitherPrint (Left e) _ = print e
```

In case parsing was successful this function applies a given function to the company and prints the result. In case of a parse error, it prints the error message. We can use this function to print the total salary of `parsedCompany` in case of parsing success:

```
1 eitherPrint parsedCompany total
```

### 2.2.7 Architecture

Parser.hs provides the Parsec-based parser. Company.hs holds the algebraic datatype for companies, while Total.hs and Cut.hs provide functionality to total and cut companies. SampleCompany.hs holds a sample company used to be compared to a parsed sample company (hosted by sample.Company). Main.hs collects test scenarios.

### 2.2.8 Usage

- Main.hs has to be loaded into GHCi.
- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* for test automation.

## 2.3 *101implementation* haskellConcurrent

### 2.3.1 Intent

Concurrent programming in Haskell

### 2.3.2 Languages

- Haskell

### 2.3.3 Technologies

- GHCi

### 2.3.4 Features

- Company
- Total
- Cut
- Concurrency

### 2.3.5 Motivation

We make use of Haskell's concurrency support. That is, we divide computations into multiple threads and make use of synchronized variables (MVar), which are shared among different threads. Each thread totals or cuts only the salaries in a specified department; subdepartments are handled by new threads. The result of the computations (`Float` resp. `Company` values) are stored in an MVar and then collected and aggregated with other results by the "upper" thread.

### 2.3.6 Illustration

#### Concurrent cutting

We create new threads using `forkIO :: IO () → IO ThreadId` provided by Haskell's concurrency library `Control.Concurrent`. This function executes the given IO action in a new thread and returns a `ThreadId` value. On the top company level we do so for every department:

Listing 2.9: Cut.hs

```
1  cutCompany :: Company -> IO Company
2  cutCompany (Company n depts) = do
3      mvars <- forM depts $ \d -> do
4          mvar' <- newEmptyMVar
5          forkIO $ cutDept mvar' d
6          return mvar'
7      cutDepts <- takeAllMVars mvars
8      return $ Company n cutDepts
```

We iterate over the departments by making use of `forM` in line 3. For each department we create a new empty `MVar` value, which we then pass to the cut function, which we start in a new thread. We collect all `MVar` values in `mvars`. In line 7 we wait for the results of the computations. The new company is returned in line 8. Similar to this we cut departments:

Listing 2.10: Cut.hs

```
1  cutDept :: MVar Department -> Department -> IO ()
2  cutDept mvar (Department n m dus eus) = do
3      mvars <- forM dus $ \d -> do
4          mvar' <- newEmptyMVar
5          forkIO $ cutDept mvar' d
6          return mvar'
7      cutDus <- takeAllMVars mvars
8      putMVar mvar $ Department n (cutEmployee m)
9                                  (cutDus)
10                                 (map cutEmployee eus)
```

The difference to `cutCompany` is that `cutDept` puts the new department in a given `MVar` value.

The cutting of direct department employees is not performed in a new thread:

Listing 2.11: Cut.hs

```
1 cutEmployee :: Employee -> Employee
2 cutEmployee (Employee name address salary) = Employee
      name address $ salary / 2
```

#### Collecting results

Both functions `cutCompany` and `cutDept` need to wait for the child-threads to terminate. To do so we provide a function `takeAllMVars`:

Listing 2.12: Utils.hs

```
1 takeAllMVars ::  [MVar a] -> IO [a]
2 takeAllMVars = mapM takeMVar
```

This function takes all `MVar` values one by one blocking on every empty MVar.

### 2.3.7 Architecture

Total.hs and Cut.hs provide functionality for totaling and cutting salaries in a concurrent way. Utils.hs contains a function to collect content of a list of `MVar` values. The algebraic datatype for companies can be found in Company.hs. Main.hs collects test scenarios for totaling and cutting a sample company hosted by SampleCompany.hs.

### 2.3.8 Usage

- Main.hs has to be loaded into GHCi.
- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* for test automation.

### 2.3.9 Issues

- The current implementation does not address the problem of a possibly unbalanced department tree.
- The collection function for MVars blocks on every empty element. We may need a more sophisticated collection function.

## 2.4 *101implementation* dph

### 2.4.1 Intent

Exercise data parallelism in Haskell using Data Parallel Haskell

### 2.4.2 Languages

- Haskell

### 2.4.3 Technologies

- GHC (Version 7.2.1)
- GHCi (Version 7.2.1)
- DPH

### 2.4.4 Features

- Company
- Total
- Cut
- Parallelism

### 2.4.5 Motivation

We exercise data parallelism in Haskell (DPH). That is, we total and cut salaries in a company by making use of parallel arrays and DPH-specific functionality on these arrays. In this context we also demonstrate some of DPH's (current) limitations (see illustration section for details):

- The inability to mix vectorised and non-vectorised code.
- No vectorization support for user-defined types.
- A DPH-specific Prelude with specific primitive types.
- Data parallelism can only be applied to arrays.

The first two limitations force us to have both vectorised and non-vectorised modules, in which we define the company datatype. The last limitation forces us to flatten the company to a list of salary values, which are of a DPH-float type, before salaries can be totaled and cut. The resulting list of cut salary values has then to be ”reconsumed” by the company in question. These flatten/consume and other conversion computations obviously take more time than we gain by switching to parallel salary functions, but this implementation is supposed to demonstrate Haskell's data parallelism support rather than being about efficiency.

### 2.4.6 Illustration

#### Cutting in Parallel

As we mentioned in the motivation section, data parallelism can only be applied to arrays. We therefore flatten the company to become a list of salaries (see SalaryFlattener.hs for details).

**Non-vectorised code** Unfortunately the normal Prelude list type `[a]` is not supported in vectorised modules, but a special array type called `PArray a`. We therefore need a special function in a non-vectorised module for converting between `[Float]` and `PArray Float` before we can cut in parallel:

Listing 2.13: Cut.hs

```
1 cut :: Company -> Company
2 cut c = (consumeSalaries c) (toList $ cutV $ fromList $
      flattenSalaries c)
```

We first flatten the company, then convert the salary list to `PArray Float` and call the cut function `cutV` from a vectorised module. After that we convert back to `[Float]` and call `consumeSalaries` to replace all salaries in the company tree.

**Vectorised code** In the vectorised module for cutting, where we want to work data parallel, we declare:

Listing 2.14: CutV.hs

```
1 {-# LANGUAGE ParallelArrays #-}
2 {-# OPTIONS_GHC -fvectorise #-}
```

This tells GHC to vectorise this module and that this module uses parallel arrays. We then define the interface function between vectorised and non-vectorised code `cutV`:

Listing 2.15: CutV.hs

```
1 cutV :: PArray Float -> PArray Float
2 {-# NOINLINE cutV #-}
3 cutV v = toPArrayP (cutVP (fromPArrayP v))
```

This function converts from `PArray Float` to a parallel array `[:Float:]`, calls the parallel code and converts back to `PArray Float`. A parallel array can only be used in a vectorised module, so only here can we convert to it. Marking this function `-# NOINLINE cutV #-` makes it usable in non-vectorised modules. `cutP` calls the actual data parallel function `cutVP`, which uses a parallel map function to cut all list values:

Listing 2.16: CutV.hs

```
1 cutVP :: [:Float:] -> [:Float:]
2 cutVP = mapP (/2)
```

### 2.4.7 Architecture

The module in SalaryFlattener.hs contains functionality to flatten a company and to replace all salaries. Total.hs and Cut.hs host the code for converting between `[Float]` and `PArray Float` and calling data parallel functions for totaling and cutting salaries, which can be found in TotalV.hs and CutV.hs. The algebraic datatype for companies can be found in Company.hs. SampleCompany.hs holds a sample company. Main.hs collects test scenarios for totaling and cutting.

### 2.4.8 Usage

- All sources have to be compiled using the GHC-options `-c -Odph -fdph-seq`.
- Main.hs has to be loaded into GHCi.
- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* for test automation.

## 2.5 *101implementation* hdbc

### 2.5.1 Intent

Database programming with HDBC

### 2.5.2 Languages

- Haskell
- SQL (MySQL dialect)

### 2.5.3 Technologies

- HDBC
- MySQL
- GHCi
- ODBC

### 2.5.4 Features

- Company
- Total
- Cut
- Persistence
- Mapping

### 2.5.5 Motivation

We use HDBC to query persisted company data. That is, we use embedded SQL in Haskell to total and cut company salaries within a database. SQL query results are mapped to special HDBC datatypes. In this context we demonstrate the use of prepared statements in HDBC. We illustrate lazy fetching of query results. To connect to the MySQL database we use an ODBC back-end. The actual functionality to cut and total salaries is independent from the concrete database implementation.

### 2.5.6 Illustration

#### Connecting

In Main.hs we connect to the MySQL database by using an ODBC driver and appropriate connection information:

Listing 2.17: Main.hs

```
let connString = "Driver={MySQL ODBC 5.1 Driver};"
               ++ "Server=localhost;"
               ++ "Port=3306;"
               ++ "Database=101companies;"
               ++ "User=root;"
conn <- connectODBC connString
```

#### Totaling

The function `total` defines a statement to total all salaries:

Listing 2.18: Total.hs

```
1  total :: IConnection conn => conn -> String -> IO
       Double
2  total conn cName = do
3       stmt <- prepare conn $
4          "SELECT salary " ++
5          "FROM employee, company " ++
6          "WHERE company.name = ? and " ++
7          "company.id = employee.cid"
8       execute stmt [toSql cName]
9       res <- fetchAllRows stmt
10      return $ sum (map (fromSql.head) res)
```

In lines 3-7 we use a prepared statement in which the company name placeholder is then replaced by the given name `cName`. The statement is executed and we use the lazy HDBC function `fetchAllRows` in line 9 to get all salaries, which we then sum up lazy to a `Double` value and return in line 10. That is, salaries are fetched one by one from the database. We can now use the open connection to total all salaries:

Listing 2.19: Main.hs

```
1 let cName = "meganalysis"
2 oldTotal <- total conn cName
```

Functionality to cut all salaries uses an UPDATE statement instead of SELECT (see Cut.hs for details).

### 2.5.7 Architecture

Company.sql and Meganalysis.sql provide SQL-scripts to create and populate company tables. Total.hs and Cut.hs provide totaling and cutting functionality using SQL statements. Main.hs collects test scenarios for totaling and cutting.

### 2.5.8 Usage

#### Setup

We need a local database server. In the following we explain the steps for XAMPP [15]. We also need an SQL tool to create and populate tables. In the following we explain the steps for the MySQL Workbench [16].

- Download and install XAMPP.
- Open the ”XAMPP Control Panel” and start ”Apache” and ”Mysql”.
- A local MySQL Server is now running:
  - **Server Host**: localhost
  - **Port**: 3306
  - **Username**: root
  - **Password**: (empty password)
- Connect to the database in MySQL Workbench.
- Select the ”101companies” schema or create it.
- Create company tables: Run the SQL script Company.sql.
- Populate company tables: Run the SQL script Meganalysis.sql.

#### Testing

- Main.hs has to be loaded into GHCi.
- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* for test automation.

## 2.6 *101implementation* haskellDB

### 2.6.1 Intent

Type-save database programming with HaskellDB

### 2.6.2 Languages

- Haskell
- SQL

### 2.6.3 Technologies

- HaskellDB
- ODBC
- HDBC
- DBDirect
- MySQL
- GHCi

### 2.6.4 Features

- Company
- Total
- Cut
- Persistence
- Mapping

### 2.6.5 Motivation

We make use of HaskellDB to express database queries as Haskell functions rather than SQL-statements (like in the *hdbc* implementation). That is, we use the rich combinator library of HaskellDB to express totaling and cutting statements based on relational algebra. We illustrate combinators for projection, selection, aggregation and renaming. Query results are mapped to values of user-defined attributes in Haskell. We connect to the underlining MySQL database through a HaskellDB-HDBC-ODBC back-end. We show the common approach of separating query/statement definition from the actual database implementation-dependent query/statement execution [17]. That is, the queries and statements themselves are database implementation-independent.

This implementation also demonstrates the use of DBDirect. We use this tool to generate modules describing the database. These modules are the basis for querying the relational data.

### 2.6.6 Illustration

#### Connecting to the database

We provide a function to connect to a database and execute an action:

Listing 2.20: MyConnection.hs

```
execute :: (Database -> IO a) -> IO a
execute = connect driver conf
  where
    conf = [ ("Driver","MySQL ODBC 5.1 Driver")
           , ("Port","3306")
           , ("Server", "localhost")
           , ("User", "root")
           , ("Database", "101companies") ]
```

We use `connect` and `driver`, which are both provided by the HDBC-ODBC back-end. By looking at the return type of the function one can see that any database function of type `Database → IO a` can be applied to `execute` resulting in the specified IO action and possibly a result of type `a`. By encapsulating the connection process like this we achieve complete independence from the underling database implementation for all queries and statements.

#### DBDirect

We use the DBDirect command `dbdirect-hdbc-odbc` (see the usage section for the complete command). This command generates a module describing the database by naming tables and fields. Compiling this module with GHC creates one module per table, each module holding actual variables for tables and fields. These variables are the basis for the following totaling query.

#### Totaling

We import the description modules for the companies and employees tables:

Listing 2.21: Total.hs

```
import qualified DBDesc.Employee as E
import qualified DBDesc.Company as C
```

We define a special field for storing the sum of all salaries:

Listing 2.22: Total.hs

```
data Ttl = Ttl

instance FieldTag Ttl where fieldName _ = "ttl"

```

```
5 ttl :: Attr Ttl Double
6 ttl = mkAttr Ttl
```

We declare `Ttl` to be an instance of the `FieldTag` class by specifying what the name of the field should be. We use this field and HaskellDB's `mkAttr` to define an attribute `ttl` for holding a `Double` value. The actual total query is defined as follows:

Listing 2.23: Total.hs

```
1  total :: String -> Query (Rel (RecCons Ttl (Expr Double
       ) RecNil))
2  total cname = do
3    es <- table E.employee
4    cs <- table C.company
5
6    restrict $
7      ( fromNull (constant 0) (cs!C.xid) .==. es!E.cid
8        .&&.
9        cs!C.name .==. constant cname )
10
11   project (ttl << _sum (es!E.salary))
```

We are working in the `Query` monad. The `table` functions return all records in the given table. Using two tables gives us the relational cross product of those tables lines 3 and 4. We use HaskellDB's selection function `restrict` in line 6-9 to select only those rows in which the company-id of the employee is equal to the company which has the given name `cname`. By making use of `project` in line 11 we only select the salary column and then use the aggregation function `_sum` to total all salaries. After that we put the total value in `ttl`.

### Executing the query

We use `query` and `execute` to execute the totaling query:

Listing 2.24: Main.hs

```
1 let cname = "meganalysis"
2 [res] <- execute $ (flip $ query) $ total cname
```

This gives us a list (which we expect to be a singleton list) of records. We can now access the `ttl` attribute of the record `res` by using the `(!)`-operator and print the total value:

Listing 2.25: Main.hs

```
1 print $ res!ttl
```

### 2.6.7 Architecture

We provide MySQL-scripts to create (see Company.sql) company tables and populate (see Meganalysis.sql) these tables. Total.hs and Cut.hs provide totaling and cutting functionality using HaskellDB's relational algebra library. MyConnection.hs encapsulates the process of connecting to the MySQL database. Main.hs collects test scenarios for totaling and cutting.

### 2.6.8 Usage

#### Setup

- Follow the steps of setting up the database as described in the usage section for the *hdbc* implementation.

#### Generating the database description

- Execute the following command in the implementation folder:

```
1 dbdirect-hdbc-odbc "DBDesc" "DBDesc" \
2                    "Driver=MySQL ODBC 5.1 Driver;\
3                    Port=3306;\
4                    Server=localhost;\
5                    User=root;\
6                    Database=101companies"
```

- Compile the *DBDesc* module using GHC: `ghc DBDesc`

#### Testing

- Main.hs has to be loaded into GHCi.
- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* covering both database descriptions generation and testing.

## 2.7 *101implementation* hxt

### 2.7.1 Intent

Tree-based XML processing with HXT in Haskell

### 2.7.2 Languages

- XML
- Haskell

### 2.7.3 Technologies

- HXT: Haskell XML Toolbox
- GHCi

### 2.7.4 Features

- Company
- Total
- Cut
- Serialization
- Mapping

### 2.7.5 Motivation

Companies are represented in XML and the Haskell XML Toolbox is used for processing such company XML data. That is, we use HXT's rich combinator library to formulate a query for totaling and a transformation for cutting salaries in a given company XML tree. Totaling results are mapped to `Float` values in Haskell. The concept of arrows is demonstrated in this implementation, because the combinator library is heavily based on this concept. That is, we exercise arrow combinators and functions.

### 2.7.6 Illustration

In the following we will demonstrate the construction of an arrow for totaling and how one can run this arrow in IO.

#### Total

We define a query for totaling all company salaries:

Listing 2.26: Total.hs

```
1 total :: ArrowXml a => a XmlTree Float
2 total = listA (deep $ hasName "salary"
3                 >>>
4                 getChildren
5                 >>>
6                 getText)
7          >>>
8          arr (sum.(map read))
```

In line 2 we query all salary nodes by using `deep $ hasName "salary" :: ArrowXml a ⇒ a XmlTree XmlTree`. This is an arrow from `XmlTree` to `XmlTree`, say a filter for all salary nodes. In general `deep` only finds non-nested results, but because of the fact that we are dealing with text nodes, which can not be nested, this is acceptable in this situation. In lines 2-6 the result of this arrow is then combined with `getChildren >>> getText` by using `>>>`. The new arrow of type `ArrowXml a ⇒ a XmlTree String` returns the text of each salary node.

We then use *Control.Arrow.ArrowList*'s `listA` in line 2 to collect all results from this arrow in an array, giving us a new arrow of type `ArrowXml a ⇒ a XmlTree [String]`.

In line 7 the result of this arrow is passed to the lifted version of `(sum.(map read)) :: (Read c, Num c) ⇒ [String] → c`, which in this case is of type: `ArrowXml a ⇒ a [String] Float`.

The overall emerging arrow, giving us the total salary, is of type `ArrowXml a ⇒ a XmlTree Float`.

### Running an arrow

When we combinate the arrow for reading a sample company from a XML file with the totaling arrow we get:

Listing 2.27: Main.hs

```
1 readDocument [] "sampleCompany.xml" >>> total
```

We use `runX :: IOSArrow XmlTree c → IO [c]` for running this arrow in IO. The function returns all results of a given arrow in a list. Because we except this list to be a singleton list, we can write:

Listing 2.28: Main.hs

```
1 [ttl] <- runX ( readDocument [] "sampleCompany.xml"
2                 >>> total )
```

`ttl` holds the total salary of a sample company.

## 2.7.7 Architecture

Total.hs provides the arrow for totaling salaries as described in the illustration section. Cut.hs contains a transformation arrow for cutting salaries. Main.hs collects test scenarios for totaling and cutting XML data provided by sampleCompany.xml

## 2.7.8 Usage

- Main.hs has to be loaded into GHCi.
- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* for test automation.

## 2.8 *101implementation* hxtPickler

### 2.8.1 Intent

H/X mapping with XML picklers

### 2.8.2 Languages

- XML
- Haskell

### 2.8.3 Technologies

- HXT
- XML pickler (comes with HXT)
- GHCi

### 2.8.4 Features

- Company
- Total
- Cut
- Serialization
- Mapping

### 2.8.5 Motivation

We exercise mapping from Haskell to XML data by making use of HXT's XML pickler arrows and functions. That is, we declare `XMLPickler` instances for companies, departments and employees and define appropriate pickler functions. In this context we illustrate predefined picklers and pickler combinators. This enables us to serialize values of algebraic datatypes for companies as XML data.

### 2.8.6 Illustration

#### A Pickler for Companies

To define a pickler for companies we declare an instance of `XMLPickler`:

Listing 2.29: Pickler.hs

```
1 instance XmlPickler Company where
2     xpickle = xpCompany
```

The pickler function `xpCompany` is defined as follows:

Listing 2.30: Pickler.hs

```
1  xpCompany :: PU Company
2  xpCompany
3        = xpElem "company" $
4          xpWrap ( uncurry Company
5                 , \c -> ( cname c
6                         , depts c
7                         )
8                 ) $
9          xpPair  (xpAttr "name" xpText)
10                 (xpList xpickle)
```

For implementing `xpCompany` we use `xpElem` in line 3. By using this function we define that a company, represented in XML, should be inside a XML tag labeled "company". The pickler for the content of the tag is defined by the second argument of `xpElem`.

`xpWrap` is of type `(a → b, b → a) → PU a → PU b`. It returns a Pickler (PU) for `b` and expects a pair of functions from `a` to `b` and vice versa and a Pickler for `a` (`PA a`).

In lines 4-8 the first part of the first argument of the wrapping pickler is `uncurry Company`. It defines how to construct a `Company` value from a pair of name and departments. The second part of the pair defines the opposite direction: How to disassemble a company into its components.

The second argument of `xpWrap` in lines 9-10 defines the actual pickler for the (name, departments) pair (this is `PA a` in the type signature of the wrapping pickler). We use the combinator for pairs `xpPair :: PU a → PU b → PU (a, b)`. The pickler for the company name is defined by using a pickler for XML attributes and a pickler for text. That is, the company name should be an attribute of the "company" tag. The pickler for the list of departments is defined by making use of a combinator for lists and `xpickle`. Because of type inference and because we also declare a `XMLPickler` instance for departments, Haskell will choose the appropriate pickler function for departments.

### Pickling a Company

Pickling a company is realized by using arrows:

Listing 2.31: Main.hs

```
1 runX ( constA company
2        >>>
3        xpickleDocument xpCompany [withIndent yes] $
```

```
4           "sampleCompanyCut.xml"
5       )
```

A lifted sample company is passed to the arrow for pickling a document. In this case `xpickleDocument` expects a pickler for companies, some writing options and a file name.

#### Unpickling a Company

To unpickle a company the arrow function `xunpickleDocument` is used:

Listing 2.32: Main.hs

```
1 [company1] <- runX ( xunpickleDocument xpCompany $
2                        [withRemoveWS yes] $
3                        "sampleCompany.xml" )
```

`runX` returns a list of arrow results, which we except to be a singleton list. On success `company1` should hold the unpicklered company.

### 2.8.7 Architecture

Pickler.hs holds the pickler definitions for companies, departments and employees. The algebraic datatype for companies can be found in Company.hs.Total.hs and Cut.hs provide totaling and cutting functionality. sampleCompany.xml holds a sample company. Main.hs collects test scenarios for pickling/unpickling, totaling and cutting companies.

### 2.8.8 Usage

- Main.hs has to be loaded into GHCi.
- The `main` function has to be applied.
- The output should be equal to the content of the file baseline.

One can also use the Makefile with a target *test* for test automation.

## 2.9 *101implementation* wxHaskell

### 2.9.1 Intent

Provide interaction on companies by means of wxHaskell

### 2.9.2 Languages

- Haskell

### 2.9.3 Technologies

- wxHaskell
- GHC
- GHCi

### 2.9.4 Features

- Company
- Total
- Cut
- Interaction

### 2.9.5 Motivation

We use the wxHaskell library to provide a simple GUI for companies. One can navigate the hierarchical company structure, cut totaled salaries and edit fields for values of primitive types. We make use of the Zipper inspired focus concept to specify which part of the company should be shown or saved after editing. This concept also helps us to read departments/employees from a given position within the company and write transformed departments/employees back.

### 2.9.6 Illustration

#### Focus datatype

We are using a datastructure inspired by the concept of Zippers to specify positions of components within the company. We provide an algebraic datatype `Focus`:

Listing 2.33: Focus.hs

```
1 data Focus =
2       CompanyFocus
3     | DeptFocus [Int]
4     | ManagerFocus [Int]
5     | EmployeeFocus [Int] Int
6      deriving (Show, Read)
```

We define one constructor per company datatype and one for managers. For example to construct a focus for an employee one needs to pass:

- A list of indexes: Starting from the company root this list is used to navigate through the departments and subdepartments to the employee's department.
- An index: The index of this employee in the employee's department's employees list.

On top of this definition we provide functions to get sub and upper foci and to read and write company components (see Focus.hs for details).

### The views

For each company datatype, that is `Company`, `Department` and `Employee`, Views.hs provides a view to display the specific component including buttons to navigate and cut salaries. All of these functions are of type `Frame () → Focus → Company → IO ()`. That is, given the global frame, a focus and a company these functions perform GUI actions (which are IO actions). In the following we demonstrate how an employee is displayed.

**Viewing an employee** `showEmployee` is the view function for employees:

Listing 2.34: Views.hs

```
1  viewEmployee :: Frame () -> Focus -> Company -> IO ()
2  viewEmployee f focus c = do
3      -- reading employee
4      let e = readEM focus c
5      -- setting up frames and panels
6      set f [ text := "Employee \"" ++ ename e ++ "\""]
7      p <- panel f [textColor := textBlue]
8      -- boxes for name, address and salary
9      nameBox <- entry p [text := ename e]
10     addressBox <- entry p [text := address e]
11     salaryBox <- entry p [text := show $ salary e]
12     -- cut button
13     cButton <- cutButton p f focus c
14     -- back button
15     bButton <- backButton p f focus c
16     -- save button
17     sButton <- button  p
18           [ text := "Save"
19           , size := Size 50 22
20           , on command := do {
21               newName <- get nameBox text;
22               newAddress <- get addressBox text;
23               newSalary <- get salaryBox text;
24               objectDelete p;
25               viewEmployee f focus $
26                writeEM focus c $
27                 Employee newName newAddress $
28                  read newSalary; }]
29     -- compose layout
30     setEmployeeLayout f p sButton bButton nameBox
          addressBox salaryBox cButton
```

In line 4 we read the employee in question using the *Focus* module's function `readEM`. In lines 6-7 we set the frame title and create a new panel for this view. In lines 9-11 we create one input box per employee field. We create buttons to cut the employee's salary and to go back to the department level in lines 13-15. In lines 17-28 we set up a save button:

When a user clicks the button, the name, address and salary fields are read, the panel is deleted, the employee is updated in the company and the new company is displayed. In the last line we call `setEmployeeLayout`, which composes the layout using various layout combinators (see Views.hs for details).

### Cutting button

On each view the GUI provides a button to cut all salaries:

Listing 2.35: Views.hs

```
1 cutButton :: Panel () -> Frame () -> Focus -> Company
     -> IO (Button ())
2 cutButton p f focus c =
3   button p [ text := "Cut"
4            , size := Size 50 22
5            , on command := do {
6                objectDelete p;
7                view f focus $ readCutWrite focus c;}]
```

When the user clicks the button the `command` action (lines 5-7) is executed. First the view's panel is deleted in line 6. We use `readCutWrite` to cut the company component's salaries. This function reads the component in question based on the focus, then cuts this component's salaries and replaces it in the given company. In line 7 the `command` action calls `view` to view the company, department or employee. Based on the focus `view` decides which concrete view-function to call:

Listing 2.36: Views.hs

```
1 view:: Frame () -> Focus -> Company -> IO ()
2 view f focus = view' f focus
3   where
4     view' = case focus of
5               CompanyFocus        -> viewCompany
6               (DeptFocus _)       -> viewDept
7               (EmployeeFocus _ _) -> viewEmployee
8               (ManagerFocus _)    -> viewEmployee
```

### Starting the GUI

We use xwHaskell's `start :: IO a → IO ()`. This function runs the given GUI of type `IO a` while discarding `a` and returning `IO ()`:

Listing 2.37: Main.hs

```
1 gui :: IO ()
2 gui = do
3     f <- frame [ textBgcolor := colorRGB 112 128 144
```

```
4                 , resizeable := False
5                 , fontWeight := WeightBold
6                 , fontUnderline := False
7                 , position := Point 50 50]
8       showCompany f CompanyFocus company
9
10  main :: IO ()
11  main = start gui
```

In lines 3-7 `gui` creates the main frame for all views passing a list attributes, which are assigned by making use of assign operator `(:=)`. In the last line the function calls `showCompany` passing the frame, the root focus and a sample company in line 8.

### 2.9.7 Architecture

Views.hs provides one view per company datatype. Main.hs starts the GUI. The algebraic datatype for companies can be found in Company.hs, a sample company in SampleCompany.hs. Focus.hs provides a focus datatype and functions on top of it. Total.hs and Cut.hs provide functionality to total and cut salaries.

### 2.9.8 Usage

- Main.hs has to be compiled using GHC
- The output executable has to run.

There is a *Makefile* with a target *start* covering both stepts.

## 2.10 *101implementation* haskellCGI

### 2.10.1 Intent

Web programming with CGI in Haskell

### 2.10.2 Languages

- Haskell
- XHTML
- CSS

### 2.10.3 Technologies

- CGI
- GHC

### 2.10.4 Features

- Company
- Total
- Cut
- Interaction
- Distribution

### 2.10.5 Motivation

We make use of the CGI library for Haskell to provide a C/S web application for companies. That is, we exercise processing requests from and returning responds to a web client using the CGI technology. We extract request parameters from the URL. That is, we make use of HTTP GET parameters. The user can either request to view or cut a company/department/employee or to save an edited company/department/employee. By making use of the Zipper inspired focus concept (see the illustration section of the *wxHaskell* implementation for details) and by passing such a focus parameter in the URL we specify which part of the company should be displayed. Client-sided company data is stored in cookies. We also demonstrate the use of a XHTML combinator library to compose new pages.

### 2.10.6 Illustration

The server is set up as follows:

Listing 2.38: Main.hs

```
1 main = runCGI $ handleErrors cgiMain
```

We use a default error handler provided by the CGI library *Network.CGI*. `cgiMain` is the main request handler, which is shown in the next section. In the following we will demonstrate how a specific request is processed by the server.

#### Cutting an employee's salary

**Scenario:** After receiving an employee view the user clicks the cut button in the browser. The browser sends a request to the server using this URL:
*http://localhost/cgi-bin/HaskellCgi/company.cgi?focus=EmployeeFocus%20[1,0,0]%200&action=Cut*

Because of the fact that the CGI library does not support any extraction of information out of the URL path (as opposed to the *happstack* implementation), parameters are encoded as URL parameters:

- The action (here `Cut`)
- The focus (here `EmployeeFocus [1,0,0]`)

**Main request handler** The main request handler is defined as follows:

Listing 2.39: Response.hs

```
1  cgiMain :: CGI CGIResult
2  cgiMain =  do
3      f <- getInput "focus"
4      let focusP = maybe CompanyFocus read f
5      a <- getInput "action"
6      let actionP = maybe View read a
7      chtml <- (doAction actionP) focusP
8      let title = "101companies WebApp"
9      output $ renderHtml $ page title $ chtml
10       where
11         doAction ap = case ap of
12           View  -> doView
13           Cut   -> doCut
14           Save  -> doSave
```

We are working inside the `CGI` monad, which is provided by the CGI library. In lines 3 and 5 `getInput :: MonadCGI m ⇒ String → m (Maybe String)` tries to get the focus and action input parameters as `String` values, which we process in lines 4 and 6. If a parameter is set, that is, `getInput` returns `Just a`, we `read` the String to values of `Focus` and `Action`. If a parameters is not set, we use a default focus respectively a default action.

We call `doAction`, which is defined in lines 11-14. Based on the `Action` value `doAction` returns one of the action functions `doView`, `doCut` or `doSave`. In this scenario the `case` expression matches on `Cut` and `doAction` returns `doCut`. `cgiMain` applies the focus to the action function in line 7. The action function returns an `Html` value, which is used in line 9 as the content of a new page, rendered to an HTML document and returned as the `CGIResult`.

**Cutting the cookie** The function `doCut` performs the actual cut action on the company cookie:

Listing 2.40: Cut.hs

```
1  doCut :: Focus -> CGI Html
2  doCut f = do
3      c <- tryReadCCookie
4      let cutC = readCutWrite f c
5      writeCCookie cutC
6      return $ html f cutC
```

The company cookie is read using `tryReadCCookie`:

Listing 2.41: Save.hs

```
1 tryReadCCookie = liftM (fromMaybe company) $
2                  readCookie "companyCookie"
```

This function tries to read the company cookie. If the client does not have this cookie stored, the default company is returned. In line 4 `doCut` calls `readCutWrite`, which reads a company, department or employee based on the focus, cuts it and replaces it within the company (see Cut.hs for details). In line 5 the manipulated company is written back into the cookie using `writeCCookie`:

Listing 2.42: Save.hs

```
1 writeCCookie s = setCookie $
2                  newCookie "companyCookie" $
3                  show s
```

**Returning HTML** After the company data is saved in the cookie `doCut` calls `html` passing the focus and the new company:

Listing 2.43: CompanyHtml.hs

```
1 html :: Focus -> Company -> Html
2 html f = case f of
3   CompanyFocus        -> companyHtml f
4   (DeptFocus _)       -> deptHtml f
5   (EmployeeFocus _ _) -> employeeHtml f
6   (ManagerFocus _)    -> employeeHtml f
```

Based on the focus `html` calls one of the functions for composing HTML. In this scenario `case` matches on `(EmployeeFocus _ _)` and `employeeHtml` is called. This function composes HTML for the employee in question using various HTML combinators (see CompanyHtml.hs for details).

### 2.10.7 Architecture

In Main.hs the server is set up using request handlers provided by Response.hs. The save actions are performed by functionality hosted by Save.hs. An algebraic datatype for actions is definied in Types.hs. HTML pages are composed in CompanyHtml.hs. The algebraic datatype for companies can be found in Company.hs. Focus.hs provides a focus datatype and functions on top of it. A sample company can be found in SampleCompany.hs. Cut.hs and Total.hs provide cut and total functionality.

### 2.10.8 Usage

- First you need a webserver. In the following we explain the steps for XAMPP [15].
- Compile Main.hs to a CGI file using GHC:

```
ghc --make -o company.cgi Main.hs
```

- Place *company.cgi* in the folder *cgi-bin* of your XAMPP installation.
- Place style.css in the folder *htdocs* of your XAMPP installation.
- Open *http://localhost/cgi-bin/haskellCGI/company.cgi* in a web browser to see the application's root view.

Compiling the project is scripted by the *run* target in Makefile.hs.

## 2.11 *101implementation* happstack

### 2.11.1 Intent

Web programming with Happstack in Haskell

### 2.11.2 Languages

- Haskell
- JavaScript
- XHTML
- CSS

### 2.11.3 Technologies

- GHC
- Happstack
- Heist

### 2.11.4 Features

- Company
- Total
- Cut
- Interaction
- Distribution
- Validation

### 2.11.5 Motivation

This implementation provides a small C/S web application written with the help of the Happstack framework. We exercise the use of routing filters to filter requests by action: A user can either view a specific part of a company, cut a specific part or save a company component after manipulation of primitive fields. The company is stored in a client-side cookie. The part to be viewed, cut or saved is specified by making use of the Zipper inspired focus concept (see the *wxHaskell* implementation for details). We also demonstrate validation: When processing a save-request, sent by an HTML form, the server applies various validators to eventually return error messages, which will be displayed to the user in the browser. As a response the client receives HTML documents, which are composed by making use of the Heist XHTML templating engine.

### 2.11.6 Illustration

In the following we will demonstrate how a specific request is processed by the server.

#### Saving an Employee

**Scenario:** After requesting to view a manager the user manipulates the input fields and submits a request by clicking a save button. The browser sends an HTTP-request together with a company-cookie to the server. The URL looks like this:
*http://localhost:8000/Employee/Save/ManagerFocus%20[0]/?Name=Erik&Address=Utrecht&Salary=1234.0*

**Routing filter** We set up a simple HTTP server:

Listing 2.44: Main.hs

```
1 main = simpleHTTP nullConf $
2   msum [ path $ \v -> path $ \a -> path $
3            \f -> mainPart a v f
4        , serveDirectory EnableBrowsing [] "static"]
```

We specify two possible server behaviours (values of `ServerPartT`) in a list, which we then apply to the `MonadPlus`-function `msum`. This function tries to run each server until one serverstart succeeds. The first list element uses Happstack's `path` function to extract:

- The view (here `Employee`)
- The action (here `Save`)
- The focus (here `ManagerFocus [0]`)

In case the extraction fails the server falls back to being a file server in line 4. In case extraction succeeds `mainPart` is called passing the action, the view and the focus:

Listing 2.45: Serverparts.hs

```
1 mainPart :: Action -> View -> Focus -> ServerPartT IO
      Response
```

```
2 mainPart View = viewPart
3 mainPart Cut  = cutPart
4 mainPart Save = savePart
```

In this scenario `mainPart` matches on `Save` and calls `savePart` passing the view and the focus:

Listing 2.46: Serverparts.hs

**Saving**

```
1  savePart :: View -> Focus -> ServerPartT IO Response
2  savePart v f = do
3    s <- save
4    case s of
5      (Left errs) -> do
6        c <- readCCookie
7        displayPart v f c errs
8      (Right newc) -> displayPart v f newc []
9      where
10       save = case v of
11         CompanyV  -> saveCompany f
12         DeptV     -> saveDepartment f
13         EmployeeV -> saveEmployee f
```

The function starts by calling a save function, which is chosen based on the given `View` value. The `save`-functions, which are all of type `Focus → ServerPartT IO (Either [(ENames,String)] Company)` either return a list of error information or the new company. In case of errors `savePart` calls `displayPart` in line 7 passing the old company (read from the cookie) and the errors. In case of success the new company and an empty list of errors is passed to `displayPart` in line 8. In this scenario `saveEmployee` is called by `savePart`:

Listing 2.47: Save.hs

```
1  saveEmployee :: Focus -> ServerPartT IO (Either [(
       ENames,String)] Company)
2  saveEmployee f = do
3    c <- readCCookie
4    name <- look "Name"
5    address <- look "Address"
6    salary <- lookRead "Salary"
7    let newe = Employee name address salary
8    let ev = validateEmployee c f newe
9    case ev of
10     (Just errs)
11         -> return $ Left errs
12     Nothing
13         -> do
```

```
14              let newc = writeEM f c newe
15              addCookie Session $
16                (mkCookie "company" (show newc))
17              return $ Right newc
```

`saveEmployee` reads the company from a cookie and extracts the request parameters from the URL in lines 3-6. These values are used to compose the new `Employee` value in line 7. In line 8 this employee is then passed to the validation function `validateEmployee` of type `Company → Focus → a → Maybe [(ENames,String)]`. If the validation succeeds, `validateEmployee` returns `Nothing`. In this case the employee is replaced within the company, which is then re-stored in the cookie and returned by the function (lines 14-16). Otherwise `validateEmployee` returns error information, which is then also returned by `saveEmployee` in line 11.

**Validation** The validation functionality can be found in the *Validators* module:

Listing 2.48: Validators.hs

```
1 validateEmployee :: Validations Employee
2 validateEmployee c f (Employee n a s) = if null vs
3                                 then Nothing
4                                 else Just $ concat vs
5                                   where
6                                     vs = catMaybes
7                                      [ validateNA c f (n,a)
8                                      , validateSalary c f s]
```

`validateEmployee` composes two validations (see Validators.hs for details):

- `validateNA` checks whether the employee's name/address pair is unique in the company `c`.
- `validateSalary` checks two things regarding the employee's salary:
  - It checks whether by changing the salary the employee's department-manager still receives the highest salary within the department.
  - It checks whether the salary has a positive value.

In case both validations return `Nothing`, `validateEmployee` returns `Nothing`. Otherwise it returns the list of all error messages.

**Binding and Responding** The user might have tried to assign an invalid salary and an invalid name/address pair to the manager in question. Validation therefore would return error information. `savePart` would call `displayPart` passing the old company and the error messages:

Listing 2.49: Serverparts.hs

```
1 displayPart :: View -> Focus -> Company -> [(ENames,
      String)] -> ServerPart Response
```

```
2  displayPart v f c errs = do
3        td <- newTemplateDirectory' tDir $
4                eNamesBinder errs $ binder f c $
5                  emptyTemplateState tDir
6        render td (B.pack tname)
7            where
8              binder = case v of
9                CompanyV  -> companyBinder
10               DeptV     -> departmentBinder
11               EmployeeV -> employeeBinder
12                 where
13             tname = case v of
14               CompanyV  -> "company"
15               DeptV     -> "department"
16               EmployeeV -> "employee"
```

In lines 8-16 `displayPart` decides which template and which binder to apply by making use of a **`case`** expression on the given view. The binder will bind all template variables to strings or small HTML fragments (splices). After that `eNamesBinder` will bind the error messages to template variables. Both binders can be found in Binder.hs. They return a function of type `Monad m ⇒ TemplateState m → TemplateState m`. That is, binders are state transformers for templates. `displayPart` then renders the HTML document, which is sent to the client as the response in line 6.

### 2.11.7 Architecture

Main.hs holds the server using various server parts in Serverparts.hs. The actual save action is performed by functionality in Save.hs. Binder.hs contains functions to bind template variables. The validators can be found in Validators.hs using helper functions hosted by Utils.hs. The algebraic datatype for companies can be found in Company.hs, a sample company in SampleCompany.hs. Functionality to total and cut is provided by Total.hs and Cut.hs. Focus.hs provides a focus datatype and functions on top of it. Various types used by the server can be found in Types.hs. The static folder contains the sytlesheet for the application and images, while templates contains the (X)HTML templates.

### 2.11.8 Usage

- Main.hs can to be consulted with runhaskell to avoid the compilation step.

There is a Makefile with a target *run* to do this.

- Open *http://localhost:8000/Company/View/CompanyFocus* to demo, starting with the root view.

# Chapter 3

# Conclusion

In this thesis we have demonstrated a wide variety of Haskell technologies. By covering most of the features of the 101companies system's feature model we have shown that Haskell is in fact very much usable in real world applications such as database programming and GUI development. The combination of modular and concise libraries and frameworks and Haskell's elegant and expressive syntax results in readable and thus maintainable applications:

- We made use of the concept of monads to compose logs in the *haskellLogger* implementation.
- We also used monads for parsing company syntax in *haskellParser*. In combination with the functor combinator library this provided an elegant and expressive way to define parsers.
- In *haskellConcurrent* we demonstrated concurrent programming at a high level of abstraction by means of MVars.
- Data parallel programming was exercised in the *dph* implementation illustrating the use of parallel arrays and the process of vectorisation.
- Two forms of database programming were covered. In *hdbc* SQL statements were represented as strings in Haskell, while the *haskellDB* implementation made use of a rich combinator library for defining queries as Haskell functions.
- Arrows and arrow combinators enabled us to process company XML data in the *hxt* implementation.
- Mapping values of algebraic datatypes in Haskell to XML data by making use of XML picklers was exercised in the *hxtPickler* implementation.
- A GUI-application for interacting with companies was realized in *wxHaskell*, which illustrated the use of a portable GUI library for Haskell. We introduced the zipper inspired concept of foci.
- Interacting with companies by means of a web application was implemented by *haskellCGI* and *wxhappstack*. The former showing a CGI-based approach using (X)HTML combinators in Haskell. The latter illustrating a web application framework providing support for routing filters and an (X)HTML template engine.

# Appendix A

# Terms and Technologies

## A.1 Writer Monad

### A.1.1 Intent

A monad in functional programming for composing auxiliary results

### A.1.2 Discussion

In a functional programming language the Writer monad can be used to compose auxiliary results, like logs [18]. In the following we will focus on Haskell.

#### Writer newtype

The Writer `newtype` can be defined as follows [19]:

```
1 newtype Writer w a = Writer { runWriter :: (a, w) }
```

That is, `Writer w a` is just a wrapper for a pair, where `a` is the type the actual result of the computation and `w` is the type of the auxiliary result.

#### Instance declaration

The Writer monad is based on ideas of [20]. Haskell defines it as follows [19]:

```
1 instance (Monoid w) => Monad (Writer w) where
2     return x = Writer (x, mempty)
3     (Writer (x,v)) >>= f = let (Writer (y, v')) = f x
4                            in Writer (y, v `mappend` v
                                ')
```

The contraint `(Monoid w) ⇒ Monad (Writer w)` in line 1 declares: `Writer w` is an instance of the `Monad` typeclass, if `w` is an instance of the typeclass monoid. The

`return` function in line 2 returns the given value `x` as the result of the computation. The auxiliary result is given by `mempty`, which is provided by the `Monoid` typeclass.

The second function for minimal complete definition is `(>>=)`. It applies `x`, the result of the given writer, to the given function `f`. `f` returns a `Writer` value with computation result `y` and the auxiliary result `v'`. The overall result is a new `Writer` with result `y` and the composed auxiliary result.

## A.2 Monoid

### A.2.1 Intent

An algebraic structure with a neutral element and an associative operation

### A.2.2 Discussion

In abstract algebra a monoid is an algebraic structure defining a neutral element and an associative binary operation [21].

In Haskell the concept of monoids is realized by using a typeclass [22].

#### Instance declaration

To declare an instance of this typeclass in Haskell one needs to define two functions (minimal complete defintion) [22]:

- `mempty :: a` (neutral element)
- `mappend :: a → a → a` (associative binary operation)

#### Lists

The list type `[a]` is an instance of the Monoid typeclass, and can be declared as follows [23]:

```
1 instance Monoid [a] where
2   mempty = []
3   mappend = (++)
```

## A.3 *Technology* Parsec

### A.3.1 Intent

A parser combinator library in Haskell

### A.3.2 Discussion

Parsec is a combinator library in Haskell to construct parsers based on Monads [24, 25]. It allows you to combine smaller parsers to more complex parsers by making use of combinators for alternatives, sequence or option. Parsec also provides some predefined parsers, for example for parsing values of primitive types.

## A.4 Functor

### A.4.1 Intent

A mathematical concept for mapping

### A.4.2 Discussion

In mathematics functors are a type of mapping [26].

In Haskell functors are realized as a typeclass defining one function, which generalizes the `map` function for lists [27].

#### Instance declaration

Declaring an instance of this typeclass requires defining `fmap` [28]:

```
1 fmap :: Functor f => (a -> b) -> f a -> f b
```

The instance declaration for lists can be defined as follows:

```
1 instance Functor [a] where
2   fmap = map
```

#### Applicative Functors

Applicative functors enrich the normal functor typeclass by various functions. They allows one to compose computations, much like monads [29, 30].

## A.5 *Technology* DPH

### A.5.1 Intent

A GHC extension for data parallelism

### A.5.2 Discussion

This extensions allows GHC to provide support for nested data parallelism [4, 31]. DPH (Data Parallel Haskell) uses vectorised modules to encapsulate data parallel code. In these modules one can make use of parallel arrays and various data parallel operations on these arrays [4].

## A.6 Parallel array

### A.6.1 Intent

An array for data parallelism in Haskell

### A.6.2 Discussion

This data structure can be used in vectorized modules in Haskell and comes with various data parallel operations [4]. In contrast to an ordinary list `[a]` parallel arrays are denoted `[:a:]` [31].

## A.7 Vectorisation

### A.7.1 Intent

Lifting functions into vector space

### A.7.2 Discussion

Vectorising a module in Haskell lifts functions in this module into the vector space [32]. It is an essential transformation in Data Parallel Haskell. According to [3] vectorisation consists of two parts:

- Transformation of all data in parallel arrays to values of primitive types.
- Transformation of code to manipulate such data.

## A.8 MVar

### A.8.1 Intent

A thread synchronization variable in Haskell

### A.8.2 Discussion

MVars are variables for thread synchronization in Haskell being either empty or holding a value [3, 33]. The module *Control.Concurrent.MVar* provides various functions for `MVar` values [34].

## A.9 *Technology* HDBC

### A.9.1 Intent

An API (implementation) for embedded SQL programming in Haskell

### A.9.2 Discussion

HDBC enables one to express database queries as strings in Haskell and execute these queries on various database implementations [35].

## A.10 *Technology* HaskellDB

### A.10.1 Intent

A combinator library for expressing database queries in Haskell

### A.10.2 Discussion

HaskellDB allows one to declare queries based on relational algebra in a type-safe and declarative way and execute these queries on a relational database [17, 36].

## A.11 *Technology* DBDirect

### A.11.1 Intent

A program generator for database definitions in Haskell

### A.11.2 Discussion

DBDirect is used in HaskellDB to generate modules describing a running database. These modules are the basis for defining queries on this database [36].

## A.12 *Technology* HXT

### A.12.1 Intent

A toolbox for tree-based XML processing in Haskell

### A.12.2 Discussion

Haskell XML Toolkit is a set of tools for processing XML data by using arrows and arrow combinators [37]. That is, the combinators are a DSL within Haskell for XML processing.

## A.13 *Technology* XML pickler

### A.13.1 Intent

An H/X mapping technology

### A.13.2 Discussion

XML picklers come with the Haskell XML Toolbox. They allow one to ”pickle” values of Haskell algebraic datatype to XML data, and ”unpickle” this data back to Haskell values [38].Both transformations are realized by using arrows:

```
1 xunpickleDocument :: PU a -> SysConfigList -> String ->
      IOStateArrow s b a
```

That is, given a pickler `PU a` for a datatype `a`, a configuration list and a filename this function returns a stateful I/O arrow. This is an arrow from an arbitrary type `b` to the type in question `a`.

## A.14 Arrow

### A.14.1 Intent

An abstract means for describing computation composition

### A.14.2 Discussion

With Arrows, like monads, one can compose computations as defined by [39]. Arrows allow you to do so in a more general and abstract way. For instance, arrows can be independent of the input or take multiple inputs [40].

In Haskell arrows are implemented as a typeclass.

#### Instance declaration

In order to declare an instance of this typeclass one needs to define two functions. [41]:

- `Arrow a ⇒ arr :: (b → c) → a b c`: Lifts a function of type `b → c` into the arrow space.
- `Arrow a ⇒ first :: a b c → a (b, d) (c, d)`: Applies a computation only to a part of the input and copies the rest to the output. [40]

## A.15 *Technology* wxHaskell

### A.15.1 Intent

A GUI library for Haskell

### A.15.2 Discussion

The wxHaskell library is built on wxWidgets allowing one to implement portable GUI applications in Haskell [42].

## A.16 Zipper

### A.16.1 Intent

A concept of a data structure for manipulating locations within a data structure

### A.16.2 Discussion

In functional programming zippers are used to write data into locations within a given data stucture. The idea of a focus that can move left, right, up and down is used to specify locations [43]. The concept was proposed by Huet [44].

## A.17 *Technology* Happstack

### A.17.1 Intent

A framework for web programming in Haskell

### A.17.2 Discussion

Some features of Happstack are [45, 46]:

- Route filters to extract URL path components.
- An integrated HTTP-server.
- Support for most Haskell database interfaces.
- Integration of various HTML templating systems such as Heist.

## A.18 *Technology* Heist

### A.18.1 Intent

A XHTML templating engine for Haskell

### A.18.2 Discussion

With Heist one can combine templates in a flexible manner. To generate dynamic web pages one can bind strings and XHTML fragments (splices) to template variables in Haskell [47, 48].

# Bibliography


[1] Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. 101 ways to cut salaries, 2011. 10 pages Online since 28 April 2011. `http://softlang.uni-koblenz.de/101companies/cut101/`.

[2] 101companies feature model. `http://101companies.uni-koblenz.de/index.php/Category:101feature`.

[3] S.P. Jones and S. Singh. A tutorial on parallel and concurrent programming in haskell. In *Proceedings of the 6th international conference on Advanced functional programming*, pages 267–305. Springer, 2008.

[4] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 383–414, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Hackage introduction. `http://hackage.haskell.org/packages/hackage.html`.

[6] The Haskell Cabal: Common Architecture for Building Applications and Libraries. `http://www.haskell.org/cabal/`.

[7] HaskellWiki: Why Haskell matters. `http://www.haskell.org/haskellwiki/Why_Haskell_matters`.

[8] HaskellWiki: Introduction. `http://www.haskell.org/haskellwiki/Introduction`.

[9] HaskellWiki: Type inference. `http://www.haskell.org/haskellwiki/Type_inference`.

[10] HaskellWiki: IO inside. `http://www.haskell.org/haskellwiki/IO_inside`.

[11] HaskellWiki: Performance/Laziness. `http://www.haskell.org/haskellwiki/Performance/Laziness`.

[12] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real world Haskell*. O'Reilly Series. O'Reilly, 2009.

[13] 101companies: README format. `http://101companies.org/index.php/101companies:README`.

[14] 101companies project page.
`http://101companies.org/index.php/101companies:Project`.

[15] Apache friends: XAMPP.
`http://www.apachefriends.org/en/xampp.html`.

[16] MySQL Workbench.
`http://dev.mysql.com/downloads/workbench/`.

[17] B. Bringert, A. Höckersten, C. Andersson, M. Andersson, M. Bergman, V. Blomqvist, and T. Martin. Student paper: HaskellDB improved. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM, 2004.

[18] Wikipedia: Monad (functional programming).
`http://en.wikipedia.org/wiki/Monad_(functional_programming)#Writer_monad`.

[19] M. Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press Series. No Starch Press, 2011.

[20] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, 1995. Springer.

[21] Wikipedia: Monoid.
`http://en.wikipedia.org/wiki/Monoid`.

[22] Hackage: Data.Monoid.
`http://hackage.haskell.org/packages/archive/base/4.4.0.0/doc/html/Data-Monoid.html`.

[23] Wikibooks: Haskell/Monoids.
`http://en.wikibooks.org/wiki/Haskell/Monoids`.

[24] Daan Leijen. Parsec, a fast combinator parser. Technical Report 35, Department of Computer Science, University of Utrecht (RUU), October 2001.

[25] HaskellWiki: Parsec.
`http://www.haskell.org/haskellwiki/Parsec`.

[26] Wikipedia: Functor.
`http://en.wikipedia.org/wiki/Functor`.

[27] GHC Documentation: Data.Functor.
`http://haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Data-Functor.html`.

[28] Hackage: Control.Monad: Functor.
`http://hackage.haskell.org/packages/archive/base/4.4.0.0/doc/html/Control-Monad.html#t:Functor`.

[29] HaskellWiki: Applicative functor.
`http://www.haskell.org/haskellwiki/Applicative_functor`.

[30] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[31] HaskellWiki: Data Parallel Haskell.
`http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`.

[32] R. Leshchinskiy, M. Chakravarty, and G. Keller. Higher order flattening. *Computational Science–ICCS 2006*, pages 920–928, 2006.

[33] GHC Documentation: Control.Concurrent.MVar.
`http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-MVar.html`.

[34] Hackage: Control.Concurrent.MVar.
`http://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/Control-Concurrent-MVar.html`.

[35] Hackage: HDBC.
`http://hackage.haskell.org/package/HDBC`.

[36] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM.

[37] HaskellWiki: HXT.
`http://www.haskell.org/haskellwiki/HXT`.

[38] HaskellWiki: Conversion of Haskell data from/to XML.
`http://www.haskell.org/haskellwiki/HXT/Conversion_of_Haskell_data_from/to_XML`.

[39] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[40] Arrows: A General Interface to Computation.
`http://www.haskell.org/arrows/`.

[41] GHC Documentation: Control.Arrow.
`http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Arrow.html`.

[42] Daan Leijen. wxHaskell: a portable and concise GUI library for haskell. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 57–68, New York, NY, USA, 2004. ACM.

[43] HaskellWiki: Zipper.
`http://www.haskell.org/haskellwiki/Zipper`.

[44] Gérard P. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[45] Happstack website.
`http://happstack.com/index.html`.

[46] HaskellWiki: Web/Frameworks: Happstack.
`http://www.haskell.org/haskellwiki/Web/Frameworks#Happstack`.

[47] Hackage: Heist.
`http://hackage.haskell.org/package/heist`.

[48] Heist tutorial.
`http://snapframework.com/docs/tutorials/heist`.