Analyzing Helium Programs Obtained Through Logging
– The process of mining novice Haskell programs –

P. van Keeken, 0264539
pkeeken@cs.uu.nl

Master Thesis

INF/SCR-05-93

Center for Software Technology,
Department of Information and Computing Sciences,
Utrecht University,
P.O. Box 80089, 3508 TB,
Utrecht, The Netherlands.

31th October 2006

Abstract

In recent years, the Helium compiler has been used to teach functional programming at Universiteit Utrecht. This compiler is equipped with a logger, which collects information about each compilation performed by a student in one of the many computer labs. Analyzing this vast collection of loggings provides information regarding the behavior of students programming in Haskell and the performance of the Helium compiler regarding the quality of error messages. This document presents the work of collecting and analyzing the data sets.

We present our approach of collecting and analyzing the data sets, which also functions as a general framework for evaluating a software tool, like a compiler, through the use of loggings. With a custom developed tool set, including a combinator library for doing descriptive statistical analysis, we present a set of interesting case study analyses. These case studies analyze programming aspects like the change of module sizes over time, the ratio of actual code lines and documentation lines over time, the amount of type hints in the compiler feedback, the time needed to repair a type incorrect program, and the effect of type hints on the time needed to repair a type incorrect program. With our approach and analysis concepts, we are able to provide empirical evidence for claims that are hardly challenged in the research field of computer science and functional programming.

Keywords: *empirical research, software tool development, functional programming, language evaluation, compiler evaluation*

Contents

Preface

This thesis has been the final work for obtaining my Master's degree in Computing Science at Universiteit Utrecht. Many hours have been spent on analyzing Helium program loggings, examining compiler feedback, and studying statistical figures, before this document could be written. While working on this thesis, a number of people provided valuable support, advice, tips, and feedback, for which I am very grateful. Firstly, I would like to thank Jurriaan Hage for his effort in meeting with me almost weekly, proofreading this thesis, and above all, his inspiring spontaneity. His comments helped me to greatly improve this thesis. Furthermore, I would like to thank Stefan Holdermans, for proofreading and support when Jurriaan was not available, Bastiaan Heeren and Arjan van IJzendoorn, for helping me to understand the sources of Helium and the ideas for possible analyses. I would like to conclude with thanking my fellow lab students, Koen, Armijn, Joost, Gerrit, and Arie for sharing the experience called "writting your Master's thesis".

Peter van Keeken, 31th October 2006

Chapter 1

Introduction

This thesis project discusses the process of collecting and analyzing program loggings obtained by the Helium compiler, a compiler used and developed at Universiteit Utrecht. Helium is specially designed for learning the functional programming language Haskell. Much effort is spent on the error reporting facility, providing the programmer clear and understandable error messages, as well as additional hints and warnings, to guide the programmer through the process of programming.

The Helium compiler is equipped with a logger, which has resulted in a vast amount of loggings collected during several functional programming courses, given at Universiteit Utrecht. These loggings are collected for the purpose of providing empirical evidence for the claim that Helium is well suited as a compiler for learning the Haskell programming language. This thesis research describes the processes of collecting and analyzing the loggings. For this, we developed a tool set, which includes utilities for preprocessing data sets of loggings, and a statistical analysis combinator library, written in Haskell. This library is useful for constructing analyses in a modular fashion and presenting the results in tables, or graphical presentations like bar charts and box plots. Furthermore, we provide a set of case study analyses, which study claims regarding the use of Helium.

This chapter discusses our approach in dealing with the analysis of sequences of loggings. In Section 1.1 we shortly discuss the current position and motivation for doing our research. Section 1.2 discusses the contribution of our research. In Section 1.3 we present an overview of similar studies applied in the field of computer language research. We conclude with an outline of this document in Section 1.4.

1.1 Position and motivation

In the field of computing sciences, claims regarding a certain tool, approach or method is often advocated by providing argumentation based on asserted reasoning [24, 28]. Hardly ever are empirical approaches used. When using a software tool like a compiler, empirical data can be much more convincing than any other method, in proving or disproving a claim. Reasons why empirical studies are rarely applied in this field, are (among others):

- *Collecting and analyzing empirical data is tedious*
- *Results are not always applicable in a broader sense, limiting the effectiveness of an empirical research*

Having empirical evidence is regarded as a strong argument in favor of one's claims. For this reason the developers of Helium implemented a logger into the compiler. Each time when a programmer compiles a program with (a specially prepared) Helium compiler on our network, the sources and some extra data, like the students' user name, is sent to a central logging server. With this feature a large collection of over 60,000 program loggings was collected during the functional programming courses in 2002/2003, 2003/2004, and 2004/2005. The collected data sets are useful for analyzing the use of Helium by studying the input and output of the compiler, as well as how subsequent loggings are related in time and content.

1.2 Contribution

In this thesis we discuss the process of collecting and analyzing the use and performance of a compiler. We do this by analyzing loggings collected during the development of a program. Our research analyzes loggings specifically obtained from the Helium compiler. Nonetheless, we think that the concepts from our research also apply for the evaluation of a compiler through logging in general.

Our contribution is twofold, we:

- *provide tools and concepts for analyzing sequences of (Helium) loggings and presenting analysis results,*
- *analyze a set of programming aspects of (students using) Helium.*

Most of our effort is spent on developing a tool set for the analysis of loggings. With this tool set we can process the logging data sets and analyze the loggings. Analyzing the loggings requires us to the ask ourselves how this is best achieved. Obviously, this requires us to take care of practical issues, like preprocessing the logging data sets. The tool set provides support for these tasks through several utilities.

For the process of analyzing the loggings and presenting the results in informative presentations, we developed a statistical combinator library, NEON. This library provides building blocks for creating, combining, and presenting statistical analyses.

This thesis also presents concepts essential to the subject of analyzing the relation between loggings. The relation between subsequent loggings can be expressed in several ways, using the difference in time or content. These notions are referred to as logging coherence and are also applicable in the broader sense of analyzing sequences of loggings from a compiler, or similar software tool.

Using the tool set and these concepts, we provide a set of case study analyses, which study claims regarding programming aspects like average module sizes, the ratio of actual code lines and documentation lines, ratio of type hints in the compiler feedback, the time needed to repair a type incorrect program, and the effect of type hints on the time needed to repair a type incorrect program. These case studies provide a first view on the possibilities of evaluating program loggings and providing empirical validation for these claims. For the discussion of these case studies we limit ourselves to the empirical data and do not infer anything beyond the sample. More research is necessary to fully evaluate these claims in-depth.

1.2.1 Thesis deliverables

The thesis work includes, along with this thesis report, the following deliverables:

- *Preprocessed (ready to compile) Helium loggings, from the functional programming courses in 2002/2003, 2003/2004 and 2004/2005*
- *A collection of Helium compilers, including all compilers used during the functional programming courses*
- *A tool set for developing analyses and presenting analysis results*

The tool set implemented by this thesis work consists of a combinator library for constructing analysis, called NEON, and a set of utilities for managing and analyzing the loggings on a file level.

1.3 Similar studies

The analysis of Haskell programs to obtain information about how Haskell is used is still very much in its infancy, and as far as we have been able to determine, the same holds for other programming languages. In literature and on the Internet we found only a few papers that consider issues related to ours, and relatively scattered throughout time. Starting in the seventies, a number of studies considered the programming behavior for various imperative languages. Moulton and Muller investigated people programming in DITRAN, a variant of FORTRAN [17], Zelkowitz considered programs written in PL/I compiled on a mainframe at Maryland University [27], and Litecky and Davis looked at novice programmers in Cobol [15]. Van den Berg studied students learning to program Miranda [13], where M. Jadud did this for Java [12]. More recently Ryder and Thompson performed a research on the application of functional metrics [19] and the effect of these metrics on later bug fixes. We shortly discuss these studies next.

Moulton and Muller evaluated the use of a version of FORTRAN developed especially for education, and focuses mainly on characterizing the types of errors made by students (divided over 21 categories). In total they considered 5,158 programs written by 234 different students and averaging 38 statements in length. Of these 1859 had compilation errors with an average of 3.8 per program.

The study of Zelkowitz involved tracing execution runs of programs written by students, but also to discover the effects on students of having followed a structured programming course previously. For example, these tended to use more comments, and fewer gotos. Zelkowitz also measured such properties as statement complexity and the measure of interactiveness of the programs.

Litecky and Davis considered 1,000 runs from a body of 50 students and classified their mistakes into 132 types of errors. They applied this characterization to a further 1,400 runs to discover that 80% of the mistakes fall into 20% of the error classes. They further analyzed the error-proneness of these errors, by compensating for the fact that some mistakes are made more, simply because the programmer has more opportunities to do so. Interesting to note is that having compared the Cobol compiler diagnoses of the mistakes with those of a Cobol expert, it turned out that 80 percent of the compiler diagnoses were wrong.

Work was done in the early nineties at Universiteit Twente on teaching the functional programming language Miranda [13] to first-year students. The study was performed empirically by following and interviewing a subset of a group of students, enrolled in their first-year functional programming course. The outcomes are of various kinds: they identify problems when learning Miranda, they discovered interferences with a concurrent exposure to an imperative language, and they even went as far to compare problem solving abilities of students who only did the functional programming course with those who did only the imperative programming course.

A more recent study was performed by Jadud by instrumenting BlueJ (a Java programming environment) to keep track of the compilation behavior of students [12]. In his study he determines for various types of errors how often they occur. The similarity with our work is that he also considers compilation behavior as the subject for analysis. Since imperative languages usually do not have a complex type system, the focus in this work is, like its predecessors from the seventies, on syntactic errors in programming.

A recent study and quite close to "home" was reported in an article by Ryder and Thompson [19]. Here they discuss the application of metrics (defined on Haskell programs) to investigate correlations between these metrics and bug fixes made to the program at a later stage. The application of their work is to identify positions in a program that are likely to benefit most from refactoring. Their experimental data consists of two program development histories, based on commits to a CVS repository. The main problem, as they identify themselves, is the fact that they simply do not have enough experimental data to validate their conclusions. The use of a CVS repository is also limiting, because usually a program that does not compile will not be committed to the repository, and most of the information we want and can obtain from our loggings has to do with how people program, and how they handle compilation errors. On the other hand, the programs they investigate actually have quite a development history, while programming assignments done by our students are done within a limited period of time, and with a fixed goal in mind.

Remark

It is interesting to note that there is a gap of at least 25 years between the first and the most recent studies [12] and [19]. This is probably due to the fact that the early languages were using highly centralized computing facilities, like mainframes. Therefore all programs were centrally processed, making it easy to gather logging information. Over the following decades decentralization took place with the introduction of PCs, while only in the last decade or so, it is possible to easily obtain information about compilation behavior by adding this into a compiler or programming environment. Note that although the studies performed by Van den Berg and his colleagues took place around 1990, and thus fall in the middle of the hiatus, their studies were done using interviews and questionnaires, and not supported by tooling.

1.4 Outline

We start by providing an introduction to the Helium compiler in Chapter 2. In Chapter 3 we discuss the experimental context in which the loggings, used in our analyses, were collected. This information is important to be able to properly interpret the results from the case studies in Chapter 6. Chapter 4 discusses concepts and methods, like notions of coherence, used in our analyses. These concepts are independent of our specific

research setup, and not related to Helium. As such, they are useful for similar studies with other compilers or similar software tools, equipped with a logging facility.

In Chapter 5 we present our analysis tools. The analysis tools include utilities for preprocessing logging data sets and a combinator library to construct and extend analyses. We discuss in detail the implementation of this combinator library. In Chapter 6 we present a set of example research studies using the analysis tool set. These case studies provide a first view on the studied programming aspect, and deliver descriptive statistical evidence, proving or disproving a stated claim. Often, these case studies yield more research questions, requiring more analysis; we leave this for future research. We conclude this thesis in Chapter 7 by providing a general overview and a discussion of possible future research steps and future applications for our analysis tool set.

Chapter 2

Helium

Helium [25, 10] is a compiler especially designed for learning the functional programming language Haskell. The Helium compiler was developed within the Software Technology [20] group at the Department of Information and Computing Sciences of the Universiteit Utrecht in the Netherlands. Quality of the error messages has been the main concern both in the choice of the language features and in the implementation of the compiler. The goal of the Helium compiler is to let students (or anyone) learn functional programming more quickly and with more fun. A major difference with other Haskell compilers is that Helium does not fully support type classes yet[1].

This chapter discusses the relevant background information about Helium, related to this thesis. In Section 2.1 we discus Haskell, the functional language on which Helium is based. We look at the currently available Haskell compilers, and how they differ from Helium. Next, in Section 2.2, we discus the features of Helium, which enable the compiler to render clear and precise error messages. We conclude this chapter with the Helium logging facility in Section 2.3. This facility is built into Helium to evaluate the compiler and allowed us to collect the logging data sets, which we analyze in this thesis.

2.1 Haskell

Haskell is a general purpose, purely functional programming language based on the functional programming paradigm [21]. This concept of functional programming has been around for a long time. Some claim that *lambda calculus* can be considered the first form of a functional programming language, although it was not designed to be executed on a computer.

Lambda calculus is a model of computation, designed by Alonzo Church in the 1930s, and provides a formal way to describe function evaluation. During the years, several researches implemented the ideas behind lambda calculus in languages like Lisp (John McCarthy, 1950) and Scheme (Guy L. Steele and Gerald Jay Sussman, around 1980). While not a purely functional programming language, Lisp introduced most of the features now found in modern functional programming languages.

Scheme was a later attempt to simplify and improve Lisp. The language Haskell was designed in the early nineties as pure functional language to gather many ideas in functional programming research, like polymorphic typing and lazy evaluation.

The reader is expected to be familiar with Haskell. Recommended references for this subject is *The Haskell School of Expression* by Paul Hudak. Information can be found on the Internet as well [21].

2.1.1 Current Haskell compilers

There are five flavors available for working with Haskell, namely Hugs [22], GHC [16], nhc98 [23], HBI/HBC [1], and Helium [25, 10].

The Haskell Interpreter Hugs [22], is a small, portable Haskell interpreter written in C, which runs on almost any machine. It offers relatively fast compilation of programs and a reasonable execution speed, and is

[1] Overloading can be used with a compiler parameter. Only basic classes, like *Eq*, are supported.

available through an interactive interpreter. Hugs also comes with a simple graphics library. However, being an interpreter, Hugs does not nearly match the run-time performance of, for example, GHC, nhc98, or HBC. Hugs is used for learning the basics of Haskell, although the error messages are not as good compared to the Helium compiler and even bad when compared to GHC [25].

GHC [16], the Glasgow Haskell Compiler, is an open source compiler and interactive environment, developed originally at the University of Glasgow. It is a full implementation of Haskell 98 plus a wide variety of extensions and works on several platforms including Windows and most varieties of Unix. GHC has extensive optimization capabilities, including inter-module optimization. Profiling is supported, both by time/allocation and various kinds of heap profiling. GHC is designed to act as a substrate for the research work of others. Downsides of GHC as compiler are its size, complexity and memory consumption.

nhc98 [23] is a small, portable, standards-compliant Haskell 98 compiler, aiming at producing small executables that run in small amounts of memory. It produces medium-fast code, and compilation is quite fast. nhc98 comes with tool support for automatic compilation, a foreign language interfacing, heap and time profiling, tracing, and debugging. Some of its advanced kinds of heap profiles are not found in any other Haskell compiler. nhc98 is available for almost all 32-bit Unix-like platforms.

HBI and HBC [1] or also called the Chalmers' Haskell Interpreter and Compiler implements Haskell 98, as well as some extensions. It is written by Lennart Augustsson, and based on the classic LML compiler by Augustsson and Johnsson. The interpreter can also load code compiled with HBC. There has been no official release for the last few years.

Helium [25, 10] is a functional programming language, compiler and interpreter designed especially for teaching Haskell, developed at the Software Technology group [20] at the Universiteit Utrecht. Quality of the error messages has been the main concern both in the choice of the language features and in the implementation of the compiler. Helium will be discussed in more detail in Section 2.2

2.2 Helium, for learning Haskell

All Haskell compilers are able to generate error messages and warnings, when compiling a piece of program code. Van IJzendoorn et al. [10] show that almost all Haskell compilers fail at providing good and clear error messages. This is often a hurdle for freshmen in learning to program in Haskell. They are new to concepts like laziness, polymorphism, and other Haskell language features. Often these language features, for instance overloading, strongly influence the clarity of an error message.

When an error message is not understandable, the programmer may lose interest, rendering the compiler feedback useless. This also means that the programmer has to find the source of the error on his own. Often a programmer is just busy coping with the compiler, than with the language.

The aim of the Helium compiler is to provide a better learning environment for Haskell. This is achieved not by only providing high quality error messages, but also by an overall approach which is aimed at getting the best results in learning Haskell. This makes Helium a unique effort in the functional programming world, where most research is spent on new language concepts and improved type systems. This section discusses the special features which are implemented in Helium to achieve its goal.

In the next section we give a short overview of how an error message looks like in Helium. From this we present several interesting characteristics which contribute to the quality of the output. We discuss these characteristics in the sections to follow.

2.2.1 Plain simple and clear error messages

Van IJzendoorn et al. [10] show many examples in which the error messages from Helium are far better than any of the currently available compilers[2]. An example of a Helium error message is presented in Figure 2.1. The example shows a verbose output of compiling a file with a type error. (The phases it is going through are normally not shown.) The error message is clear and information is nicely aligned, including the type signatures. The error location is exact. Not only a line number is given, but a column too. Other systems

[2]GHC has applied some improvements in the recent releases. Still the compiler produces not optimal error messages, due to the wide set of features supported by the compiler

```
Compiling group3/2005-01-07@16_13_39_137/CA1.hs
Lexing...
Parsing...
Importing...
Resolving operators...
Static checking...
Type inference directives...
Type inferencing...
(58,13): Type error in variable
expression       : foldl
  type           : (a   -> b   -> a  ) -> a       -> [b] -> a
  expected type  : ([c] -> [c] -> [c]) -> [[Int]] -> [Int]
probable fix     : use foldl1 instead

Compilation failed with 1 error
```

Figure 2.1: A verbose Helium error message presenting a type error.

usually just show a line number and sometimes this does not even point to the problem, but to the first line of the declaration containing the problem.

Based on experience from teaching functional programming in the past, an additional hint or a suggested fix is provided with the error message. A wide set of hints and warnings is provided, as described by Van IJzendoorn et al. [10] and on the Helium website [25]. The techniques used for this will be discussed in Section 2.2.3 and Section 2.2.4.

Another aspect to Helium is that it does not yet support overloading by default. This is an advantage when it comes to clear type errors, because the types are simpler. For instance, in the incorrect expression $1 + \texttt{'a'}$ the compiler can report that 'a' was expected to be an *Int*, instead of complaining about a *Num* instance for the *Char* type.

2.2.2 Modular compilation process

In contrast with many compilers, the Helium compiler is kept maintainable by having a sequential staging of the compilation process. The combination of a sequential staged compilation process and the use of advanced error analysis techniques, allows the compiler to report clear and easy to understand error messages and warnings are much clearer and easier to resolve, since they relate to a certain step or phase in the compilation process. In this section we discuss the phases of compilation.

When compiling a piece of code, the compiler goes through a lexing, parsing (or syntactic checking), importing, resolving operators, static checking, and type inference (as well as directives checking) analysis phase, before desugaring and actually generating executable code. These phases are also shown in Figure 2.1, an example showing verbose compilation output. Not all of these phases report errors. Errors related to the sourcecode are only reported in the lexing, parsing, resolving operators, static checking or type inferencing phase. These phases will therefore play an important role in the analyses for the Helium loggings.

The *lexical phase* represents the process of dividing program code strings into logical components, called tokens, based on punctuation and other characters, like brackets. This results in a tokenized sequence representation of the programming code. The Helium compiler generates errors and warnings when inconsistencies are detected based on punctuation.

The *parsing phase*, sometimes called the *syntactic analysis*, is the process of checking whether the code is well-formed according to the Haskell grammar. This phase results in a parse tree in which the syntactic structure of the program is made explicit. This parse tree is of a data type called UHA (Unified Haskell Architecture), which closely resembles the concrete syntax of Haskell. In this representation there still may be some detectable inconsistencies. These can be detected in the static checking and type checking phase, which both work on the UHA.

Importing is the phase in which all the required (and compiled) imports are combined into a single scope. Since the process of compiling imports is handled separately before the compilation of the parent file, this phase is not to throw any errors.

Resolving operators is the phase in which the use of operators is checked. An example of this kind of error is the ambiguous use of an associative operators. In practice, errors rarely occur in this phase.

Static analysis is the process of checking whether the program represented as a parse tree, fulfills the basic requirements for the used program elements. For instance, it is checked whether all used variables are defined.

The *type inferencing phase* is the process of checking whether all elements in the parse tree match with their required type. This phase results in a fully type correct parse tree, expressed in the UHA data type, which can be safely transformed into low level code.

The type inferencer of the Helium compiler has the following properties:

- Precise position information of an error is generated and type synonyms in error messages are preserved as much as possible.
- The programmer can choose the type inference strategy of his liking, like M [14] and W [2] and other greedy variants, or the unbiased type graph based implementations.
- The type graph implementations use several heuristics to decide what is the most likely source of the error.
- The type inferencing process can be influenced by external type directives. With these directives the programmer can develop his domain specific type rules for a combinator library he might be writing. In addition, he may specify that his experiences are that certain functions are often mixed up. As a result, a compiler may give the hint that (++) should be used instead of (:), because (++) happens to fit in the context. These type directives are discussed in more detail in Section 2.2.3.

In the *desugaring* and *code generation* phase, the type checked UHA is transformed into an enriched lambda calculus language, called *Core*. Core is then optimized and transformed in instruction files for the Lazy Virtual Machine (LVM), which can run on almost any platform.

2.2.3 Type inference directives

Error messages can be greatly improved by using knowledge about commonly occurring programming misunderstandings and domain knowledge from e.g. libraries. But there are serious disadvantages to incorporate this knowledge directly into the type inferencer of a compiler. The Helium compiler uses a system of external type inference directives from a separate '.type' file for enhancing the error messages by influencing the behavior of the constraint-based type inference process. With this one avoids the need to change the internals of the compiler and also provides ways to easily customize the compiler. Four advanced techniques provided by this system of type directives are described by Heeren, Hage and Swierstra [9].

The *siblings functions* technique provides the ability to relate commonly mistaken functions with a likely alternative, the so-called sibling function. These relations can be stated as directives. If the type inferencer runs into an inconsistency with a function mentioned among the directives, its sibling function(s) is tried as a fix for the inconsistency. For example, a student may confuse (:) with its sibling (++). If the sibling function solves the typing error, then the compiler will generate a hint along with the error, referring to the sibling function as possible solution.

Argument permutation is a technique which uses the reordering of function arguments to solve a type inconsistency. When an argument reordering solves a type error this is provided as a hint along with the error message.

Both techniques are very useful for improving error messages are, and provide mechanisms for generating hints. Other techniques described by Heeren, Hage and Swierstra [9] are *specialized type rules* and *phasing* which are more focused on the use of domain specific languages.

Specialized type rules provides the possibility to reformulate a type rule to make the type constraints more explicit. With each constraint a specialized error message (using attributes referring to variables, types and their position or range) can be formulated, providing the flexibility to generate error messages conceptually closer to the domain of a library or the understanding of the programmer. If an inconsistency is detected related to the type rule, the order of the constraints specify which constraint is selected as the source of the problem.

Phasing deals with the interpretation of language constructs. Some language constructs are strongly related to each other, but the abstract syntax tree, generated from those constructs can differ with the view functional

programmers have of them. In such cases generated error messages can be unclear and hard to understand. To overcome these scenarios, *phasing* offers a mechanism to influence the order in which constraints are globally solved for all type rules. With this better error messages can be reported, that fit with the view of the programmer.

As an extension to type inference directives, Heeren and Hage [7] describe also directives with the focus on improving error messages in presence of Haskell 98 type classes. This is yet not implemented in the Helium compiler, but planned as future work.

2.2.4 Type inferencing algorithms and heuristics

The Helium compiler is also equipped with several different typing inferencing algorithms and heuristics. Heeren, Hage and Swierstra [8] describe the type inferencing process in which constraints are generated, ordered and solved, using a constraint tree. A constraint tree is generated by use of bottom-up type inference rules.

Locating the source(s) of an error in such a constraint tree depends on the way the constraints are ordered and solved. Ordering is done by defining a tree walk on the constraint tree. This can be a standard preorder, postorder or more experimental ones such as a right-to-left tree walk. For solving constraints the Helium compiler provides several algorithms, greedy ones, which can act similar to algorithms like W [2] and M [14] and a more global analysis of a type graph.

The global constraint analysis provides much better error messages, because, when a type inconsistency is detected in the type graph, several heuristics are applied to find the most likely source of the error. This makes the algorithm more complex, but often results in a hint to accompany the error reports when displayed. The global type analysis is set as default for Helium but the user can override this by using command line parameters.

2.3 Helium logging facility

Helium is equipped with various techniques for delivering high quality error messages. It is likely that such techniques form an improvement for students learning to program. But to validate these techniques we need to look to the performance of such techniques in real life use of the Helium compiler. For this reason a logging facility was added to Helium, which logs all the programs compiled by a programmer. The logging facility is only available when Helium is built configured with the `--enable-logger` option.

With this logging facility we collected a vast amount of loggings from correct and incorrect programs, which are analyzed in this thesis. This section discusses the technical side of the logging facility. In Chapter 3, we describe how we used this facility and what (preprocessing) procedures were necessary to handle the loggings.

2.3.1 Logging procedure

Technically, the logging is performed by communication over a socket between the compiler and a Java server running on the network. The Java server is responsible for grouping loggings originating from the same account together on disk (this decision is based on an environment variable on the student's account, which gives some room for fake compiles). However, since we want to be able to say something about the process of programming, we need to know which compiles originate from the same programmer. The logged programs are time-stamped by the Java server so that we also know when a compile was performed.

Over the years, we have gradually expanded what is actually logged in the newer versions of Helium. To begin with, we log the name of the student, the version of the compiler, and the phase in which compilation terminated. We refer to each phase by its first letter, namely L for lexical, P for parsing, R for resolving operators, S for simple static errors, T for type error, and finally C for a correct compilation. The logging information is appended per compilation event to a logfile. Each line in this logfile resembles a logging event. An example of such a logfile is shown in Figure 2.2.

To have phase information readily available is essential, not just because the information itself is useful, but also because some analyses only apply to loggings that passed by or ended in a certain phase. For example, if we want to know how many type errors included a hint in the message (Helium sometimes gives explicit

```
group79:T:1.5.1 (Tue Mar 8 15:53:21 RST 2005)::group79/2005-04-08@09_18_57_519/Opdracht2.hs
group73:Sunco:1.5.1 (Tue Mar 8 15:53:21 RST 2005)::group73/2005-04-08@09_19_12_189/Propositielogica.hs
group73:Sunco:1.5.1 (Tue Mar 8 15:53:21 RST 2005)::group73/2005-04-08@09_19_24_368/Propositielogica.hs
group79:C:1.5.1 (Tue Mar 8 15:53:21 RST 2005)::group79/2005-04-08@09_19_27_396/Opdracht2.hs
group64:C:1.5.1 (Tue Mar 8 15:53:21 RST 2005)::group64/2005-04-08@09_19_46_853/Logica.hs
group73:Sunco:1.5.1 (Tue Mar 8 15:53:21 RST 2005)::group73/2005-04-08@09_19_50_094/Propositielogica.hs
...
```

Figure 2.2: Part of the log file from the 2004/2005 data set

suggestions how a program may be corrected), then we can easily ignore the majority of programs based on the logfile alone, because it can tell us which loggings resulted in a type error.

The information stored regarding the static phase is extended by a code string, describing the type of static error, shown to the user. This provides an easy way to sub categorize the static errors for a given analysis.

As an extra 'phase' an I is reported when an internal error occurs in the Helium compiler. This rarely happens and is mainly meant for the developers of Helium. When error are detected in the static phase, Helium also reports the static error message codes related to the detected static error codes. These codes are described in Appendix A.

Finally, we send the module itself. As from Helium 1.5 (released January 2005) also the modules it imports are sent as part of the logging. The latter is essential to be able to recompile the module easily on the server side. As an aside, note that we do not send along modules included in the distribution of Helium, and we do not log the compilation of expressions entered directly in the Helium interpreter. The location of the stored (main) module is included in the logfile.

Helium 1.6 (released early 2006) also sends along the full command-line parameters to the compiler, because some of the current parameters can influence both the phase in which compilation ends, and the error messages that the user obtains. Additionally, it also logs the `.type` files that were used during compilation.

The logging facility is described in-depth by Hage [5].

2.3.2 Practical remarks

Having collected all the programs, the next step is to preprocess the data before analyzing it. Obvious steps are anonymization, optional cleanup (e.g. finding module imports) and recompiling the loggings (with the original compiler) to check whether the compilation phases match. The preprocessing protocol applied in our research is discussed in Section 3.1.

It should be noticed that it is very simple to turn off the logging facility by the use of parameter. This is done to give the students the option to not cooperate with the logging of their compilations. It is of course a matter of ethics and good decency to notify the students about the Helium logging facility and to apply a strict regulation of anonymization on the data.

It is important to realize as well that the system is not fully water-proof, which is almost impossible to accomplish. For instance, by tempering with the environments variables, a student can easily transmit a logging under a false user name. In Chapter 3 we discuss how we deal with these and other limitations. Nonetheless we think that with the necessary care taken, this system can provide a very interesting set of loggings, regarding students programming Helium. This data can also tell us much about the performance of the compiler, through its error messages.

Chapter 3

Experimental context

Analyzing the Helium logging can reveal a lot of interesting information regarding novice programmers as well as compiler performance. But to be able to correctly design these analyses and interpret the analyses results, one must have a good overview of the background of the collected loggings. The collection of programs that we mine were collected during the years 2003 - 2005, during a functional programming course in which Haskell was taught to first-year students at Universiteit Utrecht. Each course was slightly different, having different requirements for the students. Also, do the data sets require some preprocessing. These subjects are discussed in this chapter.

In Section 3.1 we discuss the experimental situation of our research, important to be able to correctly interpret the analyses results later in this document. Next to this, we discuss the required preprocessing steps, before we are able to apply any analyses to the data sets. This also include an evaluation of the correctness of the collected loggings. We will see that more than 99% of the loggings are useable for analysis.

3.1 Experimental situation

With the use of a logging facility, implemented in the Helium compiler, we obtained three set of Helium program loggings during functional programming courses in the years 2003 - 2005. This means that we have collected programs in vivo, and not in vitro. In vivo is an experimental setup in which one records events from real life situations, contrary to in vitro, where recordings are made in a laboratory setting. In vivo yields information about actual use, but limits the control over the experimental situation (one might go as far as saying that there was not an experimental situation). This brings a number of complications and reservations that we shall discuss at the end of this section. First we will discuss the background of the different functional programming courses.

3.1.1 Course background information

Students attending a functional programming course, followed lectures, lab sessions and exercise classes during a period of seven to nine weeks. Due to changes in the academical system, the functional programming course was lengthened with two weeks from the academical year 2003 onwards, resulting in slightly more topics and slightly more complicated exercises in the courses in 2003/2004 and 2004/2005.

The lectures, usually twice a week, explained the theory of functional programming and were not compulsory, but strongly advised. In addition to the lectures, lab sessions and/or exercise classes were provided twice a week, which were not compulsory either. During the lab sessions students could work with a computer on programming exercises, while being supervised by experienced senior students or PhD students. During a exercise class they were asked to work on exercises on paper, often supervised as well. A lecture, lab session, or exercise class consisted of two academical hours. An academical hour consists of 45 minutes of the actual teaching or working and 15 minutes break, but these aspects are to be interpreted flexibly.

In the functional programming courses of 2002/2003 and 2003/2004 the lab sessions and exercise classes were combined, meaning that during the designated hours students could choose to work 'on paper' or work 'on the computer' in specially reserved rooms.

Course	2002/2003	2003/2004	2004/2005
Number of students being logged	119	143	83
ECTS	6	7.5	7.5
Exercise deadlines	28/03/03 11/04/03 25/04/03	05/03/04 26/03/04 16/04/04 11/06/04	04/03/05 (ICA) 11/03/05 (CKI) 08/04/05 (ICA) 22/04/05 (CKI) 20/05/05 (ICA)

Figure 3.1: Functional programming course information, 2003 – 2005

In the functional programming course of 2004/2005 the lab sessions and the exercise classes were strictly separated, meaning the supervision was focused on either one of the methods of study. (Nonetheless supervisors on exercise classes were sometimes asked to help a student working on their exercises in another room.)

Note that, based on the loggings, it is not possible to decide whether a student was in a classroom with a supervisor, actually being helped by a supervisor, or not supervised at all. This is a limitation in our setup, which we discuss in Section 3.1.2.

Besides the reserved hours for lab sessions and exercise classes, students could as well work in the computer labs on their own, since the department provides an open door policy for most of the computer rooms. When working with Helium, the students mostly worked on small exercises from the course literature or on one of the three graded assignments from the course. Students either worked alone or in teams. Only students who used the supplied Helium compiler at university network, were being logged every time they compiled their program. This results in a logging data set of both correct and incorrect programs.

The three graded assignments of the course were to be handed in per student in 2004/2005, or in a team of two students, in 2002/2003 and 2003/2004. For the third assignment of the courses of 2003/2004 and 2004/2005, students needed to use Hugs, or GHC. This resulted in virtually no loggings in the last few weeks of the course.

Unfortunately, some students dropped out during the course, not able to keep up with the lectured material and the required assignments. Exact information on why, how many, or when students stopped following the course is not available.

Information about the courses, including the deadlines of the graded assignments, are shown in Figure 3.1. We see that some courses had more than three deadlines. This is because some courses provided additional exercises, to correct a low grade. Also, had the course of 2004/2005 assignments for the students of cognitive artificial intelligence (CKI). These students had only two assignments throughout the course, where the computer science (ICA) students had three.

3.1.2 Setup limitations

The current set-up reveals a number of limitations, which implies that our statistical results are approximate. However, we are confident that given the large number of loggings we have obtained as we will see in the next section, these limitations hardly influence our results. In this section we provide an overview of these limitations.

Because the students programmed without constant supervision, we have no defense against fake compiles (a student compiles under somebody else's or a bogus name). Also, we have no control over when and where students program, and whether they do this in a classroom setting (with a student assistant present) or stand-alone. Students can also talk to each other, go out and drink coffee, obtain help from a student assistant, all invisible to us. Although students differ individually, they have a lot in common too: they have the same lecturer(s), they use the same books and in other ways share a background. This implies that conclusions about the analyzes we perform, should be drawn with care. It would certainly help to obtain loggings of programming done at other universities, but at this point we do not have this kind of information.

These complications are largely consequences of our set-up, and we can hardly expect to get around these without changing the setup or expanding our work. There are a number of complications that, to a large extent, can be dealt with by carefully considering the collection of loggings. Since we only log compiles done within the student network and they may do part of their work elsewhere, we have no guarantee of

continuity. Indeed, since logging may be turned off, gaps in the program development may even occur even when programming is done only on the network. However, by performing structural comparisons between subsequent programs we can detect many of these cases or their absence.

Since students, in some incarnations of the course, may often work on a program in pairs, they can choose to work on different accounts during different sessions (and thus the loggings for the program will be spread over two different 'teams'). We do know which students worked together (since they handed things in together) so this problem can be avoided by first merging loggings based on these names, before anonymizing.

Students do not always work on a single module, and it is possible that they change the name of a module during the programming process. It may even be that a student obtains a module from somewhere else, compiles and examines it, and afterwards proceeds with his own program. This kind of occurrence needs to be taken into account for some kinds of analyses. As a solution to these problems we introduce the notion of coherence in Chapter 4. Furthermore, a detailed evaluation of the loggings, discussed in Section 3.2.3, revealed a number of loggings based on source code copied from external sources. These loggings are disabled and not included in any or our analyses.

Even under these limitations, we think that our analyses can obtain a lot of useful information about how students (learn to) program in Haskell, but also what the pitfalls in the language are, and which parts of the language they use a lot or only little. The main reason is that we have a lot of loggings, and most of the limitations discussed above can be overcome to a large extent. For example, we can filter out the teams for which we have some evidence that loggings are missing. Such a filter will always be approximate but it can be expected that with the amount of loggings, deviations will not be significant.

3.2 Data Preparation

During the functional programming courses we have obtained 29,000 loggings in 2002/2003 from a total of 119 teams, 23,000 in 2003/2004 from 143 teams, 11,250 in 2004/2005 from 83 teams. For the analysis of these loggings it is interesting to reconstruct the exact error message that the students obtained. In this section we discuss the preparation steps needed to generate these compiler outputs and evaluate the correctness of this process. Furthermore, we also evaluate the data sets for any anomalies.

In Section 3.2.1 we discuss the preprocessing steps applied to the data sets of loggings. This includes the anonymization of loggings, and the reconstruction of module imports. Section 3.2.2 describes the different versions of Helium, used during the functional programming courses, from which we collected the program loggings. These compilers can be used to generate the compiler output which was provided to the programmer, when his program was compiled and logged. We study the correctness of our loggings and the effect of the preprocessing by comparing the resulted compilation phase, with the phase mentioned in the logfile. The results of this evaluation is discussed in Section 3.2.3.

3.2.1 Anonymization and cleaning

Having collected all the programs, the first issue to deal with is anonymization and clean-up. Anonymization is done since we are only interested in the recorded behavior and not as such into relating it any results back to a student. Anonymization was also promised to the students, as a respect to their privacy.

The anonymization takes place by renaming the student names under which compiles were logged by a random identifier (like group0, group1, ...). Because they are likely places for containing personal information, we proceed by removing all text inside comments and literal strings. This process is fully automated.

Logging reconstruction

Especially for the earlier collections of loggings, there were some issues such as spurious compilations and missing modules. For instance, the imported modules for the loggings of the data sets from 2002/2003 and 2003/2004 were not sent along with the source. However, we have been able to reconstruct virtually all the compiles with lacking import modules, because we could find the compiled module by looking a few compiles earlier in the sequence of loggings of that particular student. Here we could use the known phase description from the logfile, as a checksum, to verify that our modification was not too far off. Teams that had compiles that we could not recover, were removed completely from the collection. We will discuss the results of this

process in Section 3.2.3. The utilities we discuss in this section are also shortly discussed in Section 5.3.2 of Chapter 5.

As a first preprocessing step, we apply the script `addMainFileToLog.pl`. This utility adds the name of the main source file of a logging, to the overall logfile. With this, the overall logfile can be used to directly call the source code, a property we use in the implementation our analysis tool. Note that this utility only works before the missing module imports are found and copied to the appropriate loggings.

The following step is to locate the missing module imports for the loggings, that use these imports. This process is partially automated, using Perl and shell scripts. The utility `getallimports` finds all imports in a set of loggings. This script looks for source files (*.hs) in the subdirectories of the current directory, and outputs all imports for these files. Then, for each import module, the `globalFixImport.pl` script can be used to generate script, which in turn can be run, to copy the found programs, to the loggings that import these modules. This work for almost all cases, since often the imported module is compiled earlier by the user. The advantage of having a script to copy the found imports (at a later stage) is that it can be reapplied to the original data set, in case a small error is found in one of the previous processing steps.

Using this procedure greatly helps in finding the imports. Nonetheless, the process requires human interaction. There are several cases, like imports in imported modules, and mistakenly malformed imports which are to be treated manually. Whether the process is done correctly, can be checked by comparing the recompiled logging output with the contents of the original logfile. In a small percentage of loggings we were unsuccessful in finding the module imports, requiring us to leave these logging outside our analyzed set of loggings. The complete evaluation of the preprocessing and the resulted data, is discussed in Section 3.2.3.

Minor updates

The logging data sets are accompanied with an overall logfile. Some minor changes to this logfile and the data set, makes them better usable and more correct. We already applied the utility *addMainFileToLog . pl* to add the main file of a logging to the logfile, prior to the reconstruction of module imports. In this section we shortly discuss each of the optional updates for the logfile. Again, more information on the utilities is provided in Section 5.3.2, including the syntax of the utilities.

A small bug in the Java log server caused the logfile to record the wrong month in the time stamp of each logging. To correct this, the script `addMonthToLogfile.pl` can be applied. This also copies the logging directory structure. The output of this script can be redirected to a file, as a new logfile.

On Linux systems filenames are allowed to contain colons (':') on the older Windows systems this not allowed. Renaming the directory names from 2004-01-0314:36:43:195 to 2004-01-0314_36_43_195 can be done with the Perl script `rename`.

Both updates were applied on the data sets from the functional programming course in 2002/2003, 2003/2004 and 2004/2005.

3.2.2 Collecting Helium compilers

To reconstruct the exact error message that the student obtained, we also need to use the same compiler version. Collecting all the different compilers, turned out to be quite some work. Luckily, we can check whether the phase, mentioned in the logfile, matches with resulted phase when recompiling with the collected Helium compiler.

Currently, Helium has five major release versions, namely 1.0, 1.1, 1.2, 1.5 and 1.6. Three of these, were used during the functional programming courses of 2002/2003, 2003/2004 and 2004/2005, namely 1.1, 1.2, and 1.5. During a course, Helium was not replaced by another major release, only minor bug fixes were applied.

Nonetheless to be sure to have a similar compiler, we reconstructed for our research also the fixed (but not officially released) compilers. These versions were taken from CVS or SVN, the version management system of Helium. This resulted in a total of eleven (slightly) different compilers, shown in Figure 3.2. Nine flavors of these compilers were used during the functional programming courses from which the data sets are studied in this report.

To refer to a specific version of a Helium compiler, one would normally use the version number. But because the bug fixed editions of Helium 1.1 did not get a new version number, this is not possible. Therefore we chose

Original Version	Build Date	New Version No.	Source Origin
Helium 1.0	Thu Jan 9 16:54:03 2003	1.0	Website
Helium 1.1	Mon Mar 10 09:19:12 2003	1.1	Website
Helium 1.1	Mon Mar 10 09:19:12 2003	1.1A	Website
Helium 1.1	Tue Apr 1 19:06:55 2003	1.1.1	CVS/SVN
Helium 1.1	Mon Apr 7 16:57:57 2003	1.1.2	CVS/SVN
Helium 1.2	Tue Jan 27 14:56:04 WEST 2004	1.2	Website
Helium 1.2.1	Thu Feb 12 15:59:29 WEST 2004	1.2.1	CVS/SVN
Helium 1.2.2	Tue Mar 9 16:18:25 WEST 2004	1.2.2	CVS/SVN
Helium 1.5	Thu Jan 27 15:37:27 RST 2005	1.5	Website
Helium 1.5.1	Tue Mar 8 15:53:21 RST 2005	1.5.1	CVS/SVN
Helium 1.6	Fri Feb 3 14:53:36 RST 2006	*Unused*	Website

Figure 3.2: Collected Helium compilers

Data set	2002/2003	2003/2004	2004/2005
Number of collected loggings	29,433	23,112	11,256
Number of logged students	119	143	83
Improper loggings	12	11	0
Loggings with untraceable import(s)	0	1	0
Unsuccessful recompiled loggings	43	126	28
Amount of used loggings	29,380 (99,8%)	22,974 (99.4%)	11,228 (99.8%)
Amount of resulting teams	119	140	83

Figure 3.3: Results from cleaning the Helium loggings data sets

to also use the build date, since the combination of Helium version and build date is normally unique. But again there is an exception to this rule. The Helium compiler with the new version number 1.1A, the third listed in the Figure 3.2. This version is copy of Helium 1.1 (build date Mon Mar 10 09:19:12 2003). During the functional programming course of 2002/2003 (around March 13 2003) the lecture(s) discovered that the *signumFloat* function was missing in the *Prelude*, causing to throw internal errors as an error message. At that time, it was easier to add the function to the *Prelude*, without releasing a new Helium version.

Although an easy solution at the time, when analyzing the data set, it turned out that many loggings after March 13 showed behavior different from expected, compared to the logfile. As a solution to this problem, we created a new Helium version, and applied the same addition of the *signumFloat* function to the *Prelude*. By also updating the compiler version mentioned in the logfile, for all loggings obtained after March 13, we can still use these loggings. As a failsafe, we compared the phases of the logging after recompilation with the original phase. This yielded in a total of 53 cases of loggings, where other reasons for the difference were to blame.

For these reasons, we use in the rest of this document our own 'new' version numbers, as presented in the third column of Figure 3.2.

Remarks on compiler compilation from source

It was necessary to compile several compilers from source, because the binary installation was not available. All the compilers could be recompiled from source using GHC 6.2.2. However, this does require the user to change the Helium sources slightly, sometimes in non-trivial. These changes are described in Appendix B.

3.2.3 Data preparation results

The previous sections discusses the work necessary to be done, before the raw data sets can be analyzed. In this section we discuss the effects of the preprocessing steps, and study how the collected Helium versions can be used to check whether we can correctly reproduce the compiler output for the loggings. Not all loggings succeed for this test, as shown in Figure 3.3. The loggings for which we can not find a solution, are disabled by prefixing the related lines in the logfile with a hash ('#'). These lines are then later on ignored by our analyses.

A first rough scan of the data sets obtained some cases of loggings to be of improper nature, meaning they were not really the users own material (like source examples from the web) or not text files at all. We do not consider copying an (extensive) code example from the web to be serious novice programming. For this reason we disabled these loggings. The cases where examples are copied from the web only appear shortly in the logging set and are hardly edited, mostly because the Haskell source is not accepted by the Helium compiler. In total we found 12 cases of improper loggings in the 2002/2003 set. These cases were from three users, two tried to compile an advanced Haskell program and one somehow sent in binary data. In the 2003/2004 set we found 11 cases of improper logging, from two users. The complete set of those users was deleted because no other sensible compilation was present in their loggings. In the 2004/2005 set we found no cases of improper loggings.

The data sets from 2002/2003 and 2003/2004 both needed the source imports to be retrieved from earlier compiles. For 2004/2005 the imports were sent along with the source file to the logger, resulting in no loggings to be discarded, due to missing imports. The tracing of imports worked out very well for the 2002/2003, as well as the 2003/2004 set. In the 2002/2003 set one group (group33) showed to have 44 cases for which the required import module was not present in the logging sequence of that particular user. Fortunately, this appeared to be a standard pretty print library, provided with one of the assignments of that course. The students were not asked to edit this library. As solution we used the library as provided by the course and compared the outcome of each (re)compilation of these loggings with the outcome as described in the logfile. This appeared to be no different, giving us a strong belief that we could safely use the loggings with the chosen library.

In the data set of 2003/2004 one logging had an untraceable import. This logging appeared to be the first logging of the user in time in the data set. Disabling it would not change much of the recorded programming behavior of the user.

As a failsafe, we recompiled all data sets and compared the phase of the compilation outcome with phase mentioned for the logging in the original logfile. With this we found 43 logging discrepancies from 9 teams in 2002/2003, 126 logging discrepancies from 24 teams in 2003/2004, and 28 logging discrepancies in 2004/2005 from 8 teams.

Unfortunately we could not find a good reason for these discrepancies for the sets of 2002/2003 and 2003/2004. Likely are they related to the working of the logging facility and the effects of small errors in the anonymization process and the process of stripping string literals, comments and alike. For the 2004/2005 set the discrepancies are likely to be related with the use of an overloaded *Prelude.hs* (a new feature of the Helium 1.5 compiler). In the beginning of the course the Helium compiler was incorrectly installed on the network. The reasons for the other cases remain unknown. As a safety precaution all these non matching loggings were disabled.

After disabling the incorrect loggings, more than 99% of the original loggings remained useful for each data set, as shown in Figure 3.3. This are more than enough loggings, to be used for analyses.

Chapter 4

Conceptual approach

This thesis research empirically evaluates the Helium compiler, through the use of program loggings. This approach is not often practiced in the field of computer science, although the method could very well be used to evaluate other compilers, and similar software tools. In this chapter we discuss the concepts we think are generally applicable for these studies as well.

In Section 4.1 we discuss the concept which are useful to statistically analyze sets of loggings. These concepts come from the field of descriptive statistics. Section 4.2 discusses concepts for analyzing the relation between loggings. Often are aspects of programming represented in specially related sequences of loggings. For instance, one could be interested in sequences of loggings which represent the repair of a type incorrect program. The concepts discussed in Section 4.2, provide the tools to formulate such a sequence. These abstractions are also applicable for similar studies, analyzing software tools which are also equipped with a logging facility.

4.1 Descriptive statistics

Analyzing set of loggings is done by computing metrics for these loggings. For instance, one could calculate the length of the logged programs, the time between loggings, and many other values. To evaluate these values, one can calculate aggregate values, optionally by first grouping these values. Methods and techniques that can help to interpret the data and maintain overview, are available from the field of descriptive statistics[1]

Descriptive statistics is a branch of statistics that denotes the many techniques used to summarize a set of data, either with the goal of showing similarities or by showing how they differ. Descriptive statistics as such is not strictly defined. In this thesis the following topics are referred to as descriptive statistical operations, useful to be covered in a (descriptive) statistical library, which we introduce in Section 5.2:

- *Grouping of subjects into groups of related subjects. For example, we may group loggings per student or per week. If both groupings are applied, we obtain a sequence of loggings for each combination of student and week.*
- *Computing a statistical or computational characteristic of one or more subjects. For example, the average number of loggings for a student per week, or the length of logged program.*
- *Selecting individual subjects or groups of them, based on one or more computed characteristics.*
- *Presenting the results of our analyses in various ways, like bar plots or tables.*

The first three topics deal with the manipulation of data. These manipulations, being just functions, can be combined by composing both functions. For instance, one can develop a function which groups loggings per week, and a function which counts the number of loggings. Combining, both manipulations, provides the number of loggings per week. A next manipulation can select the weeks for which a minimum of loggings is available.

The last topic deals with effectively presenting the resulted data. The graphical and textual display of data is a typical end product of an analysis. Essential here is to support ways in which the library can fill in much of

[1] See http://en.wikipedia.org/wiki/Descriptive_statistics.

the details needed for such presentations automatically, like titles and axis names. This is essential, because otherwise a lot of time will be spent on presenting the results in a pleasing fashion. Time that could have been spent on implementing more analyses. However, this also puts extra constrains on the analysis manipulations. These manipulation should in turn provide (and maintain) the essential (background) information for any of the presentations. In Section 5.2 we discuss how we implemented these concepts of descriptive statistical analysis in a combinator library, using Haskell.

When analyzing the Helium loggings, we limit ourselves in this thesis to descriptive statistics, meaning we do not imply or infer anything beyond the studied sample. A next step in research would be to also include inferential statistics, inferring conclusions beyond the studied sample.

4.2 Coherence

In this section we consider a number of abstractions that we have found useful, indeed essential, for building analyses on sequences of loggings. In a sense, they can all be mapped back to concepts from the previous section and the notion of clustering in data mining, but their importance warrants a separate description.

The programs logged by the compiler originate from students. Usually, the students work in teams of two, or alone. As explained before, each logging comes with the name of the student on whose account the compilation was performed (whether he works alone or with someone else). To put the discussion on a more general footing we abstract away from this, referring to an entity of whom we have obtained loggings as a *loggee*.

Analyzing a large collection of programs is pretty easy when all one is interested in is to compute some value (metric) for (a subset of) the programs logged by the compiler, and afterwards computing some aggregate over these values (for conciseness of presentation). Examples of these are the average number of lines of code, the maximal nesting depth of lets and wheres, and the ratio of lines of comments with lines of code averaged over all loggees for each week.

Things become more complicated when one wants to consider the relation between loggings related in time or content. Here we use the term *coherence* to denote that two loggings are in some sense related. Based on the notion of coherence a sequence of subsequent loggings can be partitioned into a list of sequences by taking the reflexive and transitive closure of this coherence relation.

Two subsequent loggings are *time coherent* if they are apart at most k time units for some k, i.e., thirty minutes or 24 hours. The actual choice of k depends very much on the situation, so it is a parameter of the definition.

The most complicated notion we introduce is that of content coherence. Two subsequent loggings are *content coherent* if the contents of the logged modules are *similar* (to an extent which is a parameter of the definition). Various instantiations of the term similar are likely to be of use: the modules must be exactly the same, the modules differ in at most a single line, the modules have the same name, or the modules differ in at most one top-level definition.

4.2.1 Motivation for coherence partitioning

The need for these notions become obvious when one considers the following example. Say, we are interested in finding out how long it takes a loggee to correct a type incorrect program. In this case, it might well be that after some attempts the loggee gives up, and goes home. Two days later, he proceeds with the task, but in the meantime he probably spent only little time on the problem. Thus, it is unlikely that we want to consider these 48 hours as time spent on solving the mistake. When one considers the time to correct an error, one would like to apply these to a subdivision of the loggings, those that are time coherent for a suitable time limit. The notion of content coherence also comes into play here, because when one breaks the sequence of loggings for a loggee up into subsequences based on time, then such a subsequence may contain loggings that deal with different programming problems. Here the notion of content coherence can be used to distinguish between these different programming tasks, so that a type error in module `A.hs` is not considered 'solved' because the loggee compiles an unrelated, correct module `B.hs`.

Since our notion of coherence only considers subsequent loggings, the following situation might occur: a loggee is working on a module `A.hs` which after compilation gives a type error. He then remembers that he had a similar problem before. He moves to a different subdirectory and loads a module `B.hs` into the interpreter.

This module happens to need compilation and the compilation is logged. The loggee considers his solution in `B.hs` and goes back to `A.hs` to continue his work on it. The above notion of content coherence is not flexible enough to deal with the situation that we want to obtain a partitioning into sessions in which a loggee works on a certain module. Essentially, what we want to allow is some kind of look-ahead: two (possible non-subsequent) loggings are k-content coherent if they are content coherent and the number of loggings between the two (for this loggee) is at most k. Clearly, content coherent is the same as 0-content coherent. Although more time-consuming, this can be implemented straightforwardly. There is one detail left out: what do we do with the logging of `B.hs`? Do we drop it? Does it become part of a subsequent part of the division? In both cases it should be noted that the concat of the list of logging sequences is not equal anymore to the original sequence. Which option we choose, depends very much on the analysis being undertaken, so we leave this up to the programmer of the analysis.

4.2.2 Partitioning in traces

The final concept we introduce is that of a trace. In our situation we will often be interested in all the compiles made by the loggees for a given programming assignment. However, such an assignment usually takes more than a single programming session. Furthermore, the loggee might write and compile other programs during this period, for instance as part of exercise classes being followed. A trace refers to a sequence of loggings that deal with a single programming task: it consists of loggings that are content coherent with arbitrarily large look-ahead. In this case it often pays off to consider modules to be similar if and only if they have the same name or if they have similar contents. Even if someone takes his work home and returns later with a considerably modified version, we can still consider the subsequent compiles part of the same trace as long as he did not also change the filename. Note also that obtaining a trace does not mean that we have all the loggings for all the compiles of the program (see our discussion earlier), but having the traces is a good starting point for finding out whether this happens to be the case.

Traces can be computed by pairwise content coherence comparison of all loggings for a loggee, but it can be done more efficiently. First, compute content coherent sequences (for some notion of similarity), resulting in a list of sequences of loggings. Then we lift the notion of content coherence on loggings to sequences of loggings: two (possibly non-adjacent) sequences of loggings are content coherent if the final logging in the temporally earliest sequence is content coherent with the first logging of the (temporally) later sequence. Such content coherent sequences will be merged resulting in a shorter list of larger sequences. By repeatedly applying this operation to potentially all pairs of non-adjacent sequences of loggings, the set of traces for a given sequence of loggings can be obtained. Note that some care must be taken, because the end-result might depend on the order in which pairs of sequences are considered, and this also depends to some extent on the actual notion of similarity between loggings that is being used.

Remarks

Finally, note that all our notions operate on sequences of loggings of some loggee, and not between loggings for different loggees. Such an extension is possible, but we do not see a need for it yet. Note though that it is perfectly possible in our uncontrolled situation that loggings that are content coherent arise from different callees, simply because the students in question work from different accounts at different times. This problem can be avoided by preprocessing as discussed in Section 3.2.

Chapter 5

Tool set implementation

This chapter discusses the use and implementation of a tool set for handling and analyzing Helium loggings. As a first approach we discuss in Section 5.1 how the command set available on most operating systems can be used to analyze loggings on a file level. This approach is limited in its application and does not qualify as an easy extensible and scalable approach. In Section 5.2 we present our main contribution to the tool set, a statistical library, called NEON, which allows easy construction and extension of analyses, by using a combinator approach. This library also handles the presentation of analysis results. In Section 5.3 we present a set of utilities, developed during this research. These utilities provide support for smaller analysis tasks. For instance, we provide tools for calling the different versions of Helium in a flexible way.

5.1 Analysis using command line tools and scripting languages

An ad-hoc solution in developing analyses is to use the command set available on most operation systems in combination with a (shell) scripting language, like Bash or Perl. A popular command set found on most Linux like operating system today are GNU tools[1] [3]. Most command sets provide a set of tools for selecting, editing, or counting elements in files by using utilities like `cut`, `sed`, `wc`, and `diff` in combination with IO functionalities like piping[2]. Such utilities can in turn be used in (parameterized) scripts.

In the next section we present some analyses examples expressed using the GNU tool set. The reader is expected to be familiar with these GNU tools to fully understand the examples. These examples show the limitations of this approach. Scripting languages like Bash and Perl can be used to overcome these limitations to some extent. In our research we did not make extensive use of these scripting languages, but preferred a more advanced approach in the form of a combinator library written in Haskell, as discussed in Section 5.2.

5.1.1 GNU tool example analyses

For simply counting the number of loggings that ended in the typing phase, one can specify:

```
> grep ":T:" fp0304.log | wc -l
7337
```

The `grep` command returns the lines in which the string `:T:` is found. `wc -l` count the amount of lines returned by the previous `grep` command.

To count the number of type errors per user, we specify:

```
> grep ":T:" fp0405.log | sed "s/\(.*\)::\(.*\)/\2::\1/" | \
>                           sort | cut -d "/" -f1 | uniq -c
    102 group0
     49 group11
    111 group1
```

[1] GNU tools are part of the GNU operating system, which was launched in 1984 to develop a complete UNIX like operating system which is free software.

[2] Piping is using output of one command, script, or program as input for another (i.e., "chaining" the execution of programs).

```
  9 group12
 17 group14
  6 group15
    ...
```

Here we first rearrange the lines of the logfile using `sed`, so the reference to the source file is in front. Then by sorting these lines, we group the loggings of the same programmer. By splitting the line, using `cut`, only the user name remains in the output. The command `uniq -c` counts the number of occurrences per programmer.

To show the source content of these loggings, we specify:

```
> grep ":T:" fp0304.log | cut -d: -f7 | xargs -i cat {}
hello n = concat (replicate n "      ")
countSolutions a b c = [ ((b+sqrt(b*b-4*a*c))/2*a)
                       , ((b-sqrt(b*b-4*a*c))/2*a)
                       ]
...
```

Here we use the `cut` command selects the seventh field, separated by a colon. The command `xargs` executes a new command, which shows the source file using the `cat` command, for each line returned from the previous commands. Piping between commands like this is often problematic for large sets of data. `xargs` can handle this by chopping the data in appropriate pieces, avoiding data overflow. A command similar to `xargs` is `find`, which provides an option to formulate commands with the parameter `-exec <cmd>`.

Downside is that special escaping can be necessary to handle more complex commands. Also having the shell process the `xargs` command requires special care, like in the next example.

To count the amount of infinite-type errors, we specify:

```
> grep ":T:" fp0405.log | cut -d ":" -f7 |  \
>     xargs -i{} bash -c "cat {}.out*" | grep "infinite type" | wc -l
156
```

The result is calculated by first locating the type errors in the logfile, using a `grep`. Then `cut` is used to split the resulted lines in fields, separated by a colon and taking the seventh field, which is the reference to the source file. This in turn can be used by `xargs` to form a new command in which the Bash shell command expands the filename appropriately to the compilation output file. The compiler output is stored in the same directory as the source. The output file only differs from the source file by the file extension. From the output we select the lines mentioning an infinite-type error. Counting them, using `wc -l`, results in the total amount of infinite-type errors in the data set.

5.1.2 Scripting analyses

The previous examples of analyzing helium loggings demonstrate a rather crude approach in analyzing Helium loggings. Downsides are the low level of readability and limited support for error handling. Also, extending an analysis or combining analyses is usually not straightforward. For instance one could be interested to combine the analyses which groups the loggings per programmer and the analysis which count the amount of infinite-type errors to obtain the amount of infinite-type errors per programmer. These analyses can be implemented in a scripting language like Bash or Perl, although support for combining analyses is rather limited.

Our main approach for analyzing Helium loggings, which we discuss in the Section 5.2, does not use any of these languages, but relies on Haskell. A major reason for using Haskell is that we want easy access to Helium, in order to reuse parts of the compiler directly for some of the analyses. Also higher-order functions can be useful in composing different analyses. We limit the use of Bash and Perl to the smaller tasks of handling data sets. The scripts developed within our research are discussed in Section 5.3.

5.2 Statistical combinators for analyzing Helium loggings

This section discusses the implementation of Neon, an experimental library for analyzing Helium loggings. The library implements concepts from the field of descriptive statistics as presented in Section 4.1. For the

implementation we use the programming language Haskell. A motivation for this is, that many analyses often use refinements of simpler analyses by means of some form of composition, and this, we felt, is most easily expressed with higher-order functions and the like. The construction of modular analyses from a set of primitive analyses, classifies our library as combinator library.

Another reason for using Haskell is that we want easy access to the internals of Helium (itself implemented in Haskell), in order to reuse parts of that compiler directly for some of the analyses. For instance, to compute the maximal nesting depth of a Haskell program, we would like to use the Helium parser, preferably directly. Although the library is developed with the analysis of Helium loggings in mind, we do believe it is also usable for the analysis of quantitative statistical data in general.

In Section 5.2.1 we start by describing the essential aspects of the statistical analysis domain. For this we use the concepts introduced in Section 4.1. By using an example, we will also discuss requirements for such a statistical analysis library. In Section 5.2.2, we define an analysis type, which will be used by most of the analysis combinators. These combinators will be discussed in the next sections. Section 5.2.3 discusses the primitive analysis combinators, the building blocks for constructing elementary analyses. In Section 5.2.4 we discuss the more general combinators, useful for combining (primitive) analyses. Using the presented framework of constructing analyses shows some downsides, for which we present two levels of specializations in Section 5.2.5. In Section 5.2.6 we focus on the presentation of analysis results. We conclude this section with two example analyses using our library, in Section 5.2.7.

5.2.1 Statistical analysis domain description

The domain of descriptive statistical analysis is not strictly defined, which provides the room for a flexible interpretation. In this thesis, we use the description from Section 4.1 as guideline to a simple and effective implementation of a statistical analysis library.

Section 4.1 introduces four main descriptive statistical concepts, namely:

- *Grouping of subjects into groups of related subjects. For example, we may group loggings per student or per week. If both groupings are applied, we obtain a sequence of loggings for each combination of student and week.*
- *Computing a statistical or computational characteristic of one or more subjects. For example, the average number of loggings for a student per week, or the length of logged program.*
- *Selecting individual subjects or groups of them, based on one or more computed characteristics.*
- *Presenting the results of our analyses in various ways, like bar plots or tables.*

In our thesis, the subjects of study are Helium loggings. Three of these concepts, the grouping of subjects, the computation of a statistical value or a computational characteristic (or attribute) from a subject, and the selecting (or filtering) of subjects, are manipulations of data. These manipulations, can be quite easily defined with basic Haskell functions. Using function composition, a more complex computation can be constructed. The presentation concept is a typical final step in the procedure, which results in an end product of the analysis, say a table or plot of some kind.

The idea of a combinator framework is to define basic cases, the analysis primitives, for each of the three data manipulating concepts. These primitive analyses can then be used by other functions, combinators, which use one or more primitives to construct a more complex analysis. As such, an analysis developer could use this framework to construct statistical analyses, in a modular fashion. Each analysis, either a primitive analysis or a composed analysis, can be used as a building block for other analyses. As a final step, a presentation function could be applied to the result of such an analysis. However, this presentation part puts some requirements on the combinators, which we discuss given the following example.

Example For a given set of loggings, we like to know how many logged programs end in each compilation phase, each week. We present this information in a bar chart, as shown in Figure 5.7, and a table, as shown in Figure 5.8. The required analysis steps to compute this particular piece of information is to group loggings per phase, group the resulted groups again but now per week, and finally compute the number of loggings for each group. To present the information, a presentation function for a bar chart and a table is applied, resulting in the shown presentations. These presentation functions present each number of compiles and shows the related description of each group, a particular phase and week number.

The presented example shows several requirements for the combinators, mostly related to the presentation part of an analysis. The presentation of analysis results involves often not only the (numeric) outcome, but also the description of applied groupings are important. In the example, the numeric outcomes were not much of use, without the description of the related compilation phase and the number of the week. These pieces of additional information need to be included in the analysis results by the grouping analysis. Furthermore, other combined (primitive) analyses also need to correctly handle and maintain this information.

Another requirement, or actually a convenience when working with analyses, is that whenever an analysis is changed, this is also automatically reflected to the presentation of the analysis results. For instance, when one would change the presented example in computing the number of compiles per day, instead of per week, the title would change with it, without any extra human interaction. This conveniently would save time for an analysis developer. This feature, requires that all (important) primitive analyses add a description to the analysis result. These descriptions are later used by the presentation function. Whenever another analysis is using in a combined analysis, the presentation would automatically change with it.

The following requirements that are of a more general nature, are concerned with the layout of presentations displaying similar analysis results. Typically, one would like to have a similar layout for similar analyses. For instance, computing the number of compiles per phase and per week for our three loggings sets, would preferable deliver us presentations in which the weeks are always ordered chronologically, and the phases are ordered, based on their order in the compilation process, (from lexical, parsing, etc to code generation). So the ordering of phases is based on a preset order. In the rest of this document we will refer to this requirement as the ordering of (grouped) values.

Another convenient layout feature is the completion of missing values. It can happen that not all groups which like to present, are available in the analysis result. To deliver a proper presentation, these values have to be added to the analysis results before rendering a presentation, by a presentation function. For instance, the user would like to group loggings per week, in a period of week two to week nine. Unfortunately there were no loggings in week nine, due to the fact that the university was closed. Grouping the loggings, only returns the loggings from week two to nine. The analysis does not automatically now it should include this group. Solving this particular requirement, without putting too much constraints on our framework, showed to be less-trivial.

In the following sections we discuss in detail the implementation of the statistical library, and present solutions for most of these requirements. Some of the requirements remain only partially solved. For these cases the analysis developer has to provide a solution of his own.

5.2.2 Analysis type

The analysis type forms an essential part of our analysis library. This data type supports the previously discussed statistical transformations and presentation step. The presentation of computed values, as well as the grouping of data, puts extra constraints on the analysis type, as we have seem in Section 5.2.1. Often additional information, about groupings and applied analyses must be maintained. As a solution for this, we tuple the outcome of a basic computation with a description, a so-called analysis key. The purpose of this key is to maintain additional information, necessary for presentation rendering. This gives us the following type for an analysis result:

type $AnalysisResult\ key\ value = (key, value)$

In the previous section we also see the need to apply the data manipulation operations to each other's outcome, which in turn results in a new analysis. This requires an analysis to have input similar to its output. This also allows us to use function composition to implement an analysis composition. An analysis, computing a value of type b from a value of type a, then has the following type:

type $Analysis\ keya\ a\ keyb\ b = (keya, a) \rightarrow (keyb, b)$

With this type we can describe the concept of calculating a statistical value or a characteristic over the input. Unfortunately this type is not fitted for grouping. Grouping is an often occurring operation which complicates implementation of analyses. A grouping function, which is typically of type $[a] \rightarrow [[a]]$, would lead to an analysis of type:

type $GroupingAnalysis\ keya\ a\ keyb = (keya, [a]) \rightarrow (keyb, [[a]])$

This type requires the key in the analysis outcome to describe each subgroup in the outcome. For example when grouping on students, a value of type *keyb* should contain information on which students are available in the value of type $[[a]]$. If we now apply another analysis on this result, like a selection, this would require to update the key with correct information on the remaining groups. This is not the simple solution we are looking for. Therefore we rather combine a key per group, making it easier to handle the different groups. This is expressed in the type:

type *Analysis keya a keyb b* = $[(keya, a)] \rightarrow [(keyb, b)]$

In this type, we can express the basic (statistical) computations from a value of type a to type b in singleton lists. If there are more values in the analysis input, available from prior grouping, the computation is mapped over these values. A grouping with this type, is done by computing the groups of each value of type a and flattening this list into of key-value pairs of type $[(keyb, b)]$. Each key (optionally) describes the group with which the key is tupled in the analysis result. This analysis type also allows us to apply grouping analyses multiple times. In this case another grouping analysis can be applied on each group (element) of the already grouped input. The key is (optionally) updated as well, to reflect also the new grouping.

For now we leave the key data type unspecified, because this strongly depends on the presentations one is interested to have and the work the analysis developer is willing to deliver. Our framework provides a set of keys, which provide a set of presentations as we will discuss in Section 5.2.6. The more advanced presentations use the key also to cope with the requirements of ordering (grouped) values, and the completion of missing values. To prevent us in restraining the analysis developer too much, we leave the key implementation open, providing maximum flexibility.

Note For reasons of simplicity we use lists and tuples to handle analysis results, however, these data types are rather inefficient in performance. For a better performance the *Data.Map* data type can be used as well, which is part of the Haskell Hierarchical Libraries [6].

5.2.3 Primitive analysis combinators

The domain description describes a set of elementary statistical operations. In this section we discuss the primitives of our library for creating analyses, representing these elementary operations. Each primitive deals with a single concept of either calculating a characteristic, grouping values, or selecting values. These primitives return an analysis of the type discussed in the previous section.

To construct such a primitive analysis, one can choose from a range of variants, each providing a different level of generality to fit for certain needs. In this section, we first discuss the typical variant of each analysis operation, including an implementation. After this, we discuss some variants of these primitives that may be of use for non-typical analysis cases. In fact, these variants are all specializations of a more generic primitive of that particular statistical operation. As such, the primitives are also implemented in our library as specializations of the more generic primitive, unlike presented in this section. However, the functionality of the presented analysis primitives remain the same.

For the basic operation of calculating a new value from previously computed values, we introduce two combinators, *basicAnalysis* and *aggregateGrpsAnalysis*. The *basicAnalysis* combinator creates a primitive analysis which computes a new value from a single previously computed value.

basicAnalysis :: $(keya \rightarrow keyb) \rightarrow (a \rightarrow b) \rightarrow$ *Analysis keya a keyb b*
basicAnalysis keyfun valfun = *map* $(\lambda(key, value) \rightarrow (keyfun\ key, valfun\ value))$

This primitive, like most primitives, has two arguments, a key transformation function and a value transformation function. The first argument, the key transformation function, specifies how a key value, describing a value of type a, is transformed into a key value, describing the result of a computation. This computation of a value of type a into a value of type b is, not surprisingly, handled by the second argument. Both arguments are applied on a list of key-value tuples from the input of the analysis.

Example To count the number of loggings from a sequence of loggings, define:

```
countNumberOfLoggings =
  basicAnalysis (++"; Number of loggings") length
```

In this example, the key type is of type *String* and the description simply appends a piece of text to describe the operation that is performed, here computing the length of a sequence of loggings.

The *basicAnalysis* combinator only works on the values of type a in the input. If this is a list, functions like filter or aggregation functions can be used. To calculate an aggregate value over all values present in the analysis input, is a different analysis case. To handle this case, the library offers the *aggregateGrpsAnalysis* combinator.

```
aggregateGrpsAnalysis :: ([a] → [keya] → keyb) → ([a] → b) → Analysis keya a keyb b
aggregateGrpsAnalysis keyfun aggregatefun =
  (λinput →
    let
      keysin = map fst input
      valuesin = map snd input
      aggr_value = aggregatefun valuesin
    in
      [(keyfun valuesin keysin, aggr_value)]
  )
```

This combinator computes an aggregate value from all values of type a in the analysis input, using the function provided as second argument. The analysis returns a single outcome in the analysis result. The first argument transforms the keys related to the input values, into a single new key, reflecting the aggregation. However, unlike the *basicAnalysis* primitive, this time the key transformation function works on a list of values and a list of original keys. This allows us to refer to the used values in the description of this analysis.

Example To calculate the average number of loggings per student in a data set, define:

```
avgNumOfLogsPerStudent = aggregateGrpsAnalysis (++"; overall average") average
average :: Num a ⇒ [a] → Float
average = ...
```

Again we use *String* as the key type.

To specify a grouping analysis, we define the *groupAnalysis* combinator. Grouping analyses are typically of the form:

```
groupAnalysis :: (a → key1 → key2) → ([a] → [[a]]) → Analysis key1 [a] key2 [a]
groupAnalysis keyfun grpfun = concatMap groupAna
  where
    groupAna (orikey, list) = zip (map (flip keyfun orikey . head) grps) grps
      where
        grps = grpfun list
```

Implementation is different compared to the *basicAnalysis* combinator. For this primitive, the result of applying the value transformation function (the second argument) is a list of lists. The result of this function is flattened, using a *concatMap* and returned as the result of the analysis. In this case the key transformation function is a bit more complicated as well. We now not only pass the old key (describing the computations done so far), but also the first element of each computed group. From the latter value we are able to extract the information that describes the complete group, because the values are grouped on the same property. For instance, if we group loggings per week, the actual week number can be calculated from the first logging for each group of loggings. So for a grouping analysis, it is rather important to also reflect this value in the newly computed key.

Example To group loggings in sequences of loggings per student, define:

```
groupPerStudent = groupAnalysis (λlog → (++"; Grp Student " ++ (show $ username log)))
                                 groupPerStudent_
groupPerStudent_ :: [Logging] → [[Logging]]
groupPerStudent_ = ...    -- actual grouping per student
username :: Logging → String
username = ...
```

To select certain groups from the input of an analysis, the *selectGrpAnalysis* can be used. This combinator has the following implementation:

```
selectGrpAnalysis :: (key1 → key2) → (a → Bool) → Analysis key1 a key2 a
```

```
selectGrpAnalysis keyfun predfun =
  map (λ(orikey, value) → (keyfun orikey, value))
      .
  filter (λ(orikey, value) → predfun value)
```

This combinator selects the groups that fulfill the predicate (second argument), and is rather like HAVING in `SQL`. The combinator is implemented using the standard *filter* function, and applies the key transformation function to the key of the remaining groups. With this, the old key can reflect which filter has been applied.

Example To filter out all groups with less then 10 elements define:

```
selectLargerGrps = selectGrpAnalysis (++"; min. set of 10 elms") ((>10) . length)
```

Primitive variants

The primitive combinators presented so far are suitable to construct most elementary analyses. But in some cases, the description of an analysis requires a slightly a more or less general approach. For these cases, the library provides variants of the primitives, having either a more general or a less general key transformation function.

For instance, for the basic analysis primitive the library also provides a variant which includes the input value as an argument to the key transformation function. Storing this information in the key could be of use when presenting the data. This variant is of type:

```
basicAnalysis' :: (a → keya → keyb) → (a → b) → Analysis keya a keyb b
```

In the case of the grouping analysis, we also provide a version which passes the whole list of values, instead of just the first element.

```
groupAnalysis'' :: ([[a]] → keya → [keyb]) → ([a] → [[a]]) → Analysis keya [a] keyb [a]
```

Example The variant *groupAnalysis''* can be used to define an analysis, which groups a sequence of loggings into groups of time coherent sequences as described in Section 4.2. Typically, you would like to label each session with a number. This number is not computable from a single logging of a session, but from the complete grouped set.

```
groupPerSession = groupAnalysis''
                    (λgrps oldkey → map (((oldkey ++ "; " ++ "Session: ")++) . show) [0..length grps])
                    groupPerSession_
  where
    groupPerSession_ :: [Logging] → [[Logging]]
    groupPerSession_ = ...    -- actual session grouping
```

The library provides for each analysis primitive a set of variants, suited for particular cases. A complete overview of the available combinator variants is presented in Appendix D. All variants are basically more general or less general versions of the primitives presented in the first part of this section. As such, they are also defined in the more generic counter part in our library.

5.2.4 General combinators

The previous section discusses the primitive combinators, which describe the basic cases or building blocks of an analysis. In this section we present combinators which combine these analyses with the purpose to construct larger, more complex analyses.

To compose two analyses, the <·> combinator can be used.

```
(<·>) :: Analysis keyb b keyc c → Analysis keya a keyb b → Analysis keya a keyc c
(<·>) = (.)
```

The <·> combinator is of similar type like function composition ((.) :: $(b \to c) \to (a \to b) \to a \to c$), and is implemented in terms of it. Combined analyses using the <·> combinator are applied from right to left, similar to function composition.

Example To build an analysis which first groups per session and then calculated the number of loggings, define:

```
countSessionsSize = countNumberOfLoggings <·> groupPerSession
```

For completeness we also provide a combinator for the application operator, which is also implemented in terms of it.

```
(<$>) :: Analysis keya a keyb b → [(keya, a)] → [(keyb, b)]
(<$>) = ($)
```

The *mapAnalysis* combinator lifts an analysis computing a value of type b from value of type a, to lists of these types. Often an analysis is easy to define for a single subject. This combinator can be used to construct a similar analysis which works on a list of subjects. This combinator is implemented by:

```
mapAnalysis :: Analysis keya a keyb b → Analysis keya [a] keyb [b]
mapAnalysis ana =
  map (λ(key, value) →
    (head $ (getKeyTransf ana) key, concat $ map (getAnalysisFun ana) value))
getKeyTransf :: Analysis keya a keyb b → (keya → [keyb])
getKeyTransf = ...
getAnalysisFun :: Analysis keya a keyb b → (a → [b])
getAnalysisFun = ...
```

Example To calculate the phase of a logging, define:

```
phaseAnalysis = basicAnalysis "phase" phase

phase :: Logging → Phase
phase = ...
```

To lift this analysis to also work for a list we define:

```
phasesAnalysis = mapAnalysis phaseAnalysis
```

The <&> operator tuples two analyses, to be applied independently to the same input. This combinator is implemented by:

```
<&> ::Analysis keya a keyb1 b1 → Analysis keya a keyb2 b2 → Analysis keya a (keyb1, keyb2) (b1, b2)
ana1 <&> ana2 = (λinput →
  let ana1result = ana1 input
      ana2result = ana2 input
  in
      zip (zip (map fst ana1result)
          (map fst ana2result)) (zip (map snd ana1result) (map snd ana2result))
  )
```

Example To calculate the longest and the shortest logged module, we define:

```
minMaxLOCAnalysis = minimumAnalysis <&> maximumAnalysis
minimumAnalysis :: Analysis keya [Int] keyb Int
minimumAnalysis = ..
maximumAnalysis :: Analysis keya [Int] keyb Int
maximumAnalysis = ..
```

To execute an analysis, the function *runAnalysis* is provided. This function runs the provided analysis with the provided key and input data.

```
runAnalysis :: a → keya → Analysis keya a keyb b → [(keyb, b)]
runAnalysis input startkey ana = ana [(startkey, input)]
```

In some cases it might be interesting to split the result from a grouping analysis, and separately apply (or map) other analyses and presentation functions on these sub-results. The *splitAnalysis* combinator can be used for this.

```
splitAnalysis :: [(key, a)] → [[(key, a)]]
splitAnalysis anaresult = [[x] | x ← anaresult]
```

This combinator simply extracts each element of the input and presents it in a list of singleton lists. Another analysis can conveniently be mapped over the resulted list. Due to this behavior, the combinator is not of the typical analysis type like the other combinators.

A variant is the *splitAnalysis′* combinator. This combinator also takes an argument, which specifies which groups from the input are put together. This variant comes in handy, when, after applying multiple groupings, one would like to have certain groups put together. In some cases, it might be easier and faster in terms of performance to regroup the obtained result using the *splitAnalysis′* combinator, than to recompute such an analysis from scratch.

```
splitAnalysis′ :: (Eq a, Eq key) ⇒ (key → key → Bool) → [(key, a)] → [[(key, a)]]
splitAnalysis′ keycompare = groupAllByEq newcmp
  where newcmp (key1, v1) (key2, v2) = keycompare key1 key2
groupAllByEq :: (a → a → Bool) → [a] → [[a]]
groupAllByEq = ...
```

All analyses can be combined using the analysis composition combinator <·>, only in some cases this does not result in the desired outcome. For instance, to count the number of compiles per phase in a set of loggings, define:

```
numberOfLoggingsPerPhase :: Analysis String [Logging] String Int
numberOfLoggingsPerPhase =
      basicAnalysis (++"; number of logs") length
  <·> groupPerPhase
```

If we want to calculate the overall average per phase one can define:

```
avgNoLogsPerPhase :: Analysis String [Logging] String (Maybe Float)
avgNoLogsPerPhase =
      averageOverall
  <·> numberOfLoggingsPerPhase

averageOverall :: Analysis String Int String (Maybe Float)
averageOverall = aggregateAnalysis (++"; average") (mean)
```

But using this analysis on another grouped analysis, say per student, does not work as expected, because the average is calculated over all the grouped sets for all students. In this case we would like to reuse the *avgNoLogsPerPhase* analysis to apply it on each of the groups of loggings per student independently. For this we introduce the *isolateAnalysis* combinator.

```
isolateAnalysis :: Analysis keya a keyb b → Analysis keya a keyb b
isolateAnalysis analysis =
      concat
  <·> map analysis
  <·> splitAnalysis
```

Example To build an analysis which calculates the average for each group of loggings per student, define:

```
avgNoLogsPerPhasePerStudent :: Analysis String [Logging] String (Maybe Float)
avgNoLogsPerPhasePerStudent =
      isolateAnalysis avgNoLogsPerPhase
  <·> groupPerStudent
```

Using the *isolateAnalysis* combinator, we can also develop a special combinator for composing analyses:

```
(<·>) :: Analysis keyb b keyc c → Analysis keya a keyb b → Analysis keya a keyc c
ana1 <·> ana2 = (isolateAnalysis ana1) <·> ana2
```

Example The analysis from the previous example can now be expressed as:

```
avgNoLogsPerPhasePerStudent' :: Analysis String [Logging] String (Maybe Float)
avgNoLogsPerPhasePerStudent' =
        avgNoLogsPerPhase
    <:> groupPerStudent
```

A combinator related to the *groupAnalysis* combinator is the *flattenAnalysis* combinator. This combinator flattens the input of the analysis of type $[(b, a)]$, by integrating the values of type b into the key of the analysis, similarly like with a grouping analysis.

```
flattenAnalysis :: ((b, a) → key1 → key2) → Analysis key1 [(b, a)] key2 a
flattenAnalysis keyfun =
        basicAnalysis id (snd . head)
    <:> groupAnalysis keyfun split

split :: [x] → [[x]]
split list = [[x] | x ← list]
```

This combinator is useful in combination with a presentation. Sometimes you would like to present using an one-dimensional presentation instead of a two-dimensional presentation.

5.2.5 Specialization of the primitive functions

The combinators introduced so far present a very open but sometimes toilsome framework for setting up analyses. One of the problems which surfaces when using the combinators, is that constructing an analysis, also requires the programmer to specify each time how the key should be transformed. For instance, when the key is of type *String*, an obvious solution would be to concatenate a string to the existing key. A downside is that, the programmer has to specify this for each analysis. As a solution, we introduce type classes, which deal with these common analysis tasks.

A first specialization is the *Keyable* type class. This class handles the often occurring process of combining keys from the input, describing the analyses done so far, with a key transformation function to a new key, describing (also) the applied analysis. By specifying also an initial key for the key data type, simplifies the *runAnalysis* combinator. The class is defined by:

```
class Keyable a where
    startkey :: a
    combine :: a → a → a
    combineList :: [a] → a → a
    combineList keys = flip (foldl combine) keys
```

The *combineList* function is for combining a set of keys. This function is applied to combine keys of an aggregating analysis. This analysis applies an aggregate function to a list of values, and also requires to combine a list of keys. As a default, the *combineList* is defined in terms of the *combine* function.

Implicit in the *Keyable* class is that the input and output key are of the same type. This results in a simplification of the analysis type.

```
type AnalysisFK key a b = Analysis key a key b
```

The primitive analysis combinators are simplified in the number of arguments.

```
basicAnalysis          :: Keyable key ⇒ key → (a → b) → AnalysisFK key a b
aggregateGrpsAnalysis  :: Keyable key ⇒ ([a] → key) → (a → b) → AnalysisFK key a a
groupAnalysis          :: Keyable key ⇒ (a → key) → ([a] → [[a]]) → AnalysisFK key [a] [a]
selectGrpAnalysis      :: Keyable key ⇒ key → (a → Bool) → AnalysisFK key a a
runAnalysis            :: Keyable key ⇒ a → AnalysisFK key a b → [(key, b)]
```

The key transformation functions (the first parameters) now do not require a function which explicitly transforms the old key into a new key. Only a key describing the applied analysis is required. This simplification also applies to the variants of the primitives, as shown in Appendix D.

An instance for a key of type *String* could very well be:

```
instance Keyable String where
  startkey = ""
  combine orikey newkey = if (null orikey) then newkey
                                           else  orikey ++ "; " ++ newkey
```

The *String* key is readable from left to right, with "; " as separator between the descriptions of the applied analyses.

Another specialization on top of the *Keyable* class encapsulates a fixed key transformation function for each primitive analysis. It is defined as:

```
class Keyable a ⇒ DescriptiveKey a where
  basicKey      :: String → a
  aggregateKey :: String → [a] → a
  groupKey      :: DataInfo b ⇒ Either b (String, String) → a
  selectGrpKey :: String → a
  tupleKey      :: a → a → a → a
```

This type class specifies the minimal information that should be provided for a key data type, to be able to render proper presentations. Each member function, except the last function, describes the required information for a particular analysis primitive. This information is used by related primitive analysis combinators, to simplify the use of the combinator.

For the basic analysis case, we require only a string, describing the computation. An aggregate analysis requires a descriptive string, and a list of values. The list of values can optionally be integrated into a resulted key. For a grouping, we either require a value for which an instance of *DataInfo* is available, or two descriptive strings. These two strings describe the property of a grouping. For instance, when grouping per student one string requires the name of the student (student "bob") and the other the description of this attribute ("students"). The presentation function is expected to be able to properly handle these values. The *DataInfo* encapsulates these same values, as we will see later in Section 5.2.6, where we discuss the presentation part of our library. This class also provides room for other presentation features. The primitive analysis for selecting values requires only a string, describing the selection.

The type class also requires a function for combining keys in a tupling analysis. The tuple combinator is actually not part of the set of primitive analysis combinators, but does require special attention when it comes to presenting the result of a tupled analysis. Therefore this member function was added to this class. The member function requires to specify how the key, provided as input to the tupling analysis, and the keys resulted after applying the tupled analyses, should be combined.

When an instance for this class is provided, we can make use of the following primitive combinators, including tupling:

```
basicAnalysis          :: DescriptiveKey key ⇒ String → (a → b) → AnalysisFK key a b
aggregateGrpsAnalysis :: DescriptiveKey key ⇒ String → ([a] → b) → AnalysisFK key a b
groupAnalysis          :: (DescriptiveKey key, DataInfo b) ⇒ (a → b) →
                                                    ([a] → [[a]]) → AnalysisFK key [a] [a]
selectGrpAnalysis      :: DescriptiveKey key ⇒ String → (a → Bool) → AnalysisFK key a a
runAnalysis            :: DescriptiveKey key ⇒ a → AnalysisFK key a b → [(key, b)]
(<&>) :: DescriptiveKey key ⇒ AnalysisFK key a b1 → AnalysisFK key a b2 → AnalysisFK key a (b1, b2)
```

The presented combinators limit the required description of each combinator to a very minimum. All analysis primitives require a descriptive string, except the grouping analysis. The grouping analysis requires the user to specify a function which calculate the property on which the subjects are grouped. For the data type of this property should an instance of *DataInfo* be available. This class can be used to provide presentation features, which we discuss in Section 5.2.6. Variants of this combinator exists where the grouping can be described using two descriptive strings. These variants, as well as the more general variants of the other analysis cases are shown in Appendix D.

Provided key definitions

The NEON library provides a set of predefined keys, for which instances of the *Keyable* class and the *DescriptiveKey* class are provided. Using these instances, also provides the use of the related, simpler analyses combinators. In the previous section, we already presented the instance of *Keyable* for a key of type *String*. Similarly, one can also define this for a *Maybe* data type.

```
instance (Keyable a) ⇒ Keyable (Maybe a) where
  startkey = Nothing
  combine Nothing newkey = newkey
  combine (Just x) newkey =
    case newkey of
      Nothing → Just x
      Just key → Just (x `combine` key)
```

This instance requires that there is also a *Keyable* instance available for the encapsulated type of *Maybe*. We also can define an instance for the *DescriptiveKey* class for the *String* data type.

```
instance DescriptiveKey String where
  basicKey descr          = "Basic analysis " ++ show descr
  aggregateKey descr keys = "Aggregated calculation: " ++ descr ++ " based on " ++ show keys
  groupKey eitherval =
    either (λvalue → "Grouping analysis " ++ (showType value) ++ ": " ++ show value ++ "")
           (λ(str1, str2) → "Grouping analysis " ++ show str1 ++ ": " ++ show str2 ++ "")
           eitherval
  selectGrpKey descr      = "Select analysis " ++ show descr
  tupleKey basekey ana1key ana2key =
    "Tupled analysis (" ++ basekey ++ ", " ++ ana1key ++ ", " ++ ana2key ++ ")"
```

But as will see in Section 5.2.6, this type is not really usable for rendering presentations. Especially, the ordering of elements is limited to only lexicographic ordering. Therefore, we provide in our library the *KeyHistory* data type, which is more suited for presentation rendering. This data type is based the *KeyHistoryDescriptor* data type and provides the structure to encapsulate every primitive analysis case, with the required content for presenting the result. In our library, we also provide a set of presentation functions, specifically for this data type. These presentations are discussed in Section 5.2.6.

The *KeyHistory* data type and the *KeyHistoryDescriptor* data type are defined as:

```
newtype KeyHistory = KHL{ getKHL :: [KeyHistoryDescriptor] }
  deriving (Eq, Ord, Show)
data KeyHistoryDescriptor where
  BasicAnalysis      :: String → KeyHistoryDescriptor
  AggregateAnalysis :: String → [KeyHistory] → KeyHistoryDescriptor
  GroupingAnalysis  :: DataInfo a ⇒ Either a (String, String) → KeyHistoryDescriptor
  SelectGrpAnalysis :: String → KeyHistoryDescriptor
  TuplingAnalysis    :: KeyHistory → KeyHistory → KeyHistory → KeyHistoryDescriptor
```

The *KeyHistory* consists of a list of *KeyHistoryDescriptor*. The *KeyHistoryDescriptor* data type defines an existential data type [18, 11, 26] using GADT syntax. The *KeyHistoryDescriptor* provides a set of data fields. Each such field provides the structure to encapsulate the essential information of one of the analysis primitives, as discussed earlier. As we will see later, each of these fields will be used as key for the instance declaration of the *DescriptiveKey* class of the *KeyHistory* data type.

Existential data types are a new feature in GHC. It allows us to specify *KeyHistoryDescriptor*, so we now not only can store the necessary strings, as for the basic analysis and the selecting analysis, but also link an instance of *DataInfo* of a grouping property, to the key. This provides us a higher level of flexibility than just using a string as key. The *DataInfo* class will be discussed in Section 5.2.6.

With these data types, to which we also will refer in the rest of this document as simply the *KeyHistory* data type, we can provide an instance for the *Keyable* and the *DescriptiveKey* class.

```
instance Keyable KeyHistory where
```

```
  startkey = KHL []
  (KHL k1) `combine` (KHL k2) = KHL $ k1 ++ k2
instance DescriptiveKey KeyHistory where
  basicKey descr              = KHL [BasicAnalysis descr]
  aggregateKey str keys       = KHL [AggregateAnalysis str keys]
  groupKey grp                = KHL [GroupingAnalysis grp]
  selectGrpKey val str        = KHL [SelectGrpAnalysis str]
  tupleKey orikey key1 key2 = KHL [TuplingAnalysis orikey key1 key2]
```

The *Keyable* instance declaration is straightforward. Each newly applied analysis adds a key to the encapsulated list of *KeyHistoryDescriptor*. The instance declaration of *DescriptiveKey* links each field of the *KeyHistoryDescriptor* to one of the key, related to each primitive analysis. We now can use the simplified analysis combinators of the *DescriptiveKey* class. Section 5.2.6 discusses how presentation are rendered using the *KeyHistory* data type. Furthermore, Section 5.2.7 presents complete examples of the use of the analysis combinators and the presentation functions.

5.2.6 Presenting analysis results

This section discusses how we can render presentations from the result of an analysis. The result of an analysis is a list of tupled key-value pairs of type $[(key, value)]$. As such this data can be stored in a file. However, it is nicer to present the data in a table or graph. Our statistical library provides several presentation functions for this. Our approach is do this as effortlessly as possible, by using presentation functions of type:

$render1DTable :: (Ord\ key, Show\ value) \Rightarrow [(key, value)] \rightarrow MarkUpDoc$

The information stored in the key data type is used to compute the components of a presentation, like titles, column headers, and axis labels. So generating a presentation greatly depends on the used key data type. Typically, one has to write a set of presentation functions for each key type. This may sound as a great disadvantage but is a consequence of the freedom that one gets from using the analysis combinators. If presentation functions are provided for a certain key type, they can be reused easily.

This section discusses how we implemented presentation functions in our library. We chose to provide a set of presentation functions for the *KeyHistory* data type as key. The *KeyHistory* data type presents enough structure and content to present data in almost any form.

To discuss our implementation, we first present a simple approach in implementing presentation functions for the analysis results, by using *String* as type for the analysis key. This approach shows some limitations of our presentation concept. To counter these limitations, we present a solution by using the *KeyHistory* type as key. In this discussion we refer to the presentation requirements discussed in Section 5.2.1.

Next, we discuss the current implementation of the presentation functions, using the *KeyHistory* as key. This implementation is slightly different, than proposed for practical reasons. In addition, we discuss the different presentation forms, available in the library, and the underlying tool used for the actual rendering of presentations. The combined use of the analysis combinators with the presentation functionality is shown in a full blown analysis example in Section 5.2.7

Simple analysis presentation

To present data obtained from a statistical analysis one could use (custom made) presentation functions, which (in their most basic form) are of type:

$render1DTable :: String \rightarrow String \rightarrow [(String, String)] \rightarrow String$
$render2DTable :: String \rightarrow [String] \rightarrow [(String, [String])] \rightarrow String$

These particular presentation functions are capable of rendering a table, either a simple one-dimensional (a list like) table or a two-dimensional (cross-)table, by giving a title, one or more column headers (in a list) and providing the actual table data, in a list of tupled strings. Similar functions can also be provided for rendering bar charts, box plots, and other graphical presentations.

Using such presentation functions requires the analysis result to be transformed into the different parameters of these presentations. It would be convenient to have a (partially automated) way to obtain each presentation

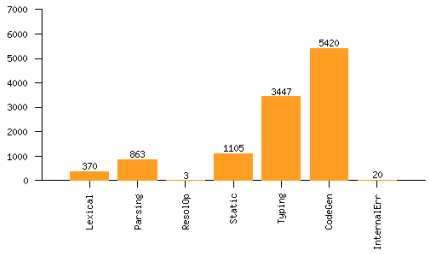


Figure 5.1: Bar char of number of loggings per phase

parameter from the analysis data. Functions responsible for such transformations strongly depend on the type of the key, and can be tricky for more complex analysis results. But once implemented, they can be reused when needed. This would result in a set of easy to use simple presentation functions, such as:

$$\begin{aligned}
&\mathit{render1DTableSmart} &&:: \mathit{Ord}\ \mathit{key} \Rightarrow [(\mathit{key}, \mathit{value})] \rightarrow \mathit{String}\\
&\mathit{render2DTableSmart} &&:: (\mathit{Ord}\ \mathit{key}, \mathit{Ord}\ a) \Rightarrow [(\mathit{key}, [(a, b)])] \rightarrow \mathit{String}\\
&\mathit{render2DTableSmart}' &&:: \mathit{Ord}\ \mathit{key} \Rightarrow [(\mathit{key}, \mathit{value})] \rightarrow \mathit{String}
\end{aligned}$$

These functions can be easily used in mapping over a list of analysis results. The *Ord* class is required, so the data can be properly ordered in the presentation. For the rendering of two-dimensional presentations, we provide two variants. The first variant, *render2DTableSmart*, lets the developer design an analysis which returns the second dimension in the value of the analysis result(by using the tupling analysis primitive).

Another approach is to design an analysis which uses the grouping combinator at least two times, and compute a result for each group. If the key correctly and distinctively administers the groupings, the function *render2DTableSmart′* should able to dissect the two dimensions of groups from the keys of the analysis result. For this operation the reusable function *ungroup* can be defined, having the type: $[(key, b)] \rightarrow \mathit{Maybe}\ [(key, [(a, b)])]$. Other two-dimensional presentation functions, like a stacked bar chart, can make use of this function as well.

By default the *ungroup* function takes the encapsulated value in the key, which describes the last applied grouping and makes the value explicit in the result. The actual type of the ungrouped a value depends on the key implementation. If the ungrouping fails, the presentation function still can render the presentation in an one-dimensional way. For instance, a normal bar chart can be rendered, if data is not properly grouped to be presented as a stacked bar chart.

The value which describes the grouping is essential for the rendering of a presentation. For instance, when the data is grouped per student, this value contains the name of the student to which the group of loggings belong. Ordering and completion of missing values must be done based on these values. In the latter of this document we refer to this value as being the *group index* of an applied grouping analysis.

In the rest of this section, we discuss the use of strings as key type, like shown in the many examples in the previous sections. This type provides an easy way to concatenate all interesting describing parts of our analysis. By using regular expressions or parsers, we can extract information for the presentations, like a title (based on the applied calculations), the group names for the column headers, and optionally sort the data lexicographically.

Example To compute the number of compiles per phase, one could use the following analysis:

$$\begin{aligned}
&\mathit{countLogsPerPhase} :: \mathit{AnalysisFK}\ \mathit{String}\ [\mathit{Logging}]\ \mathit{Int}\\
&\mathit{countLogsPerPhase} = \quad \mathit{basicAnalysis}\ \texttt{"number of loggings"}\ \mathit{length}\\
&\qquad\qquad\qquad <\cdot> \mathit{groupPerPhase}
\end{aligned}$$

The results of this analysis is shown in Figure 5.1, although this graph is not rendered using a *String* as key type. The key-value pairs of these analysis results, when using a key of type *String*, could very well be:

```
[ ("number of loggings; Group Phase: Lexical;",     370),
  ("number of loggings; Group Phase: Parsing;",     863),
  ("number of loggings; Group Phase: Static;",     1105),
  ("number of loggings; Group Phase: ResolOp;",       3),
  ("number of loggings; Group Phase: CodeGen;",    5420),
  ("number of loggings; Group Phase: Typing;",     3447),
  ("number of loggings; Group Phase: InternalErr;",  20),
```

```
]
```

The title for this presentation is obtained by analyzing the keys from the analysis result. By combining the invariant factors, like the string "number of logs" and the string "Group Phase", we obtain a title as shown in the presentation. Here we concatenate the string, using default infix words for each analysis primitive. The labels on the x-axis is obtained by taking the group index values from each key, for instance "Lexical".

Using *String* as key type has some major disadvantages. Because if the key is a string, this also holds for the group indexes. This makes it not possible to choose for another (ofter more logical) non-lexicographic order. For instance, if loggings are grouped per day (an ordinal value), and the particular day of each group is encapsulated in a string, the presentation function can not properly order these the days. Lexicographical ordering does not suit here. An additional parameter should be added to each presentation function to cope with this problem.

The presentation of analysis results also yields a problem which is of a more general nature, like we discussed shortly in Section 5.2.1. When presenting data for several similar cases, one would like to have a similar layout. For example, when presenting the average size of source files per week for a set of five students (from the same logging set), one would like to get five graphs, each presenting the same weeks. Unfortunately, in our current implementation there is no guarantee that this will always happen. When grouping data, for instance per week, the resulted data are groups of data for each week *present in the original set*. The weeks not present in the original set, are not part of the result set at all. These groups can be absent for different reasons: the related data is filtered out in a previous analysis, or the data is simply not present. Whatever reason, when presenting the results it would be convenient to have all relevant groups displayed.

To some extent, this problem could be solved when grouping is done with an enumerable and bounded data type. However, we do not want to require this, since we prefer to have an open analysis framework, not restricting the developer in its choice of data types. Furthermore, this also would not completely solve the problem, since some ordinal data are not meant to be bounded in the first place. For instance, when grouping loggings into sessions, one cannot beforehand decide how many sessions will be formed.

Dealing with these issues of ordering and missing group values makes it hard to develop an effortless model for displaying analysis result. Using a string as analysis key, is not very practical to cope with these issues. Ordering is limited to only lexicographic ordering, and completion of missing groups values must be handled by the developer of the analysis.

Enhanced analysis presentation

To cope with the issues described in the previous section, we use the *KeyHistory* data type as key, in combination with an instance for the *DescriptiveKey* class, as discussed in Section 5.2.5. In the next section, we discuss our current implementation in our library, based on this concept.

The *DescriptiveKey* class requires extra information to deal with the issues of presentation, especially regarding the group indexes. A group index, like the week number when grouping data per week, can be defined by either two strings or a value, for which we require an instance of the *DataInfo* class. These requirements are expressed by $groupKey :: DataInfo\ b \Rightarrow Either\ b\ (String, String) \rightarrow a$ member function of the *DescriptiveKey* class.

Describing a group index by two strings, is similar as describing a grouping with a string as key. One string can be used to describe (the type of) the group index, useful information to put on an axis title. The other can be used to contain the actual value, to be used as a label for the group on the axis. This approach presents the same problems regarding the ordering of analysis data, as presented in the previous section.

By describing the group index using the *DataInfo* class is more practical in handling the problems of ordering and missing values. This type class provides functions for the naming and ordering of a group index.

The *DataInfo* class is defined as:

```
class Show a ⇒ DataInfo a where
  showType  :: a → String
  showValue :: a → String
  showValue = show
  ordIndex  :: a → Int
  allvalues :: ComplementInfo a
  allvalues = NoCompletion
```

ComplementInfo is defined as:

data *ComplementInfo* $a =$
  GroupValus $[a]$
  NoCompletion

The ordering of the groups encapsulated in a key of type *KeyHistory* can be handled by using the *ordIndex* member function. We cannot use a regular $compare :: a \to a \to Ordering$ function for ordering, specified by the *Ord* class, because an existential data type does not allow to access the actual encapsulated value. As a solution, we require the group index data type to have a unique index number for each of its values. Ordering can now be done, by comparing the *ordIndex* of group indexes. The *ordIndex* is similar to the *fromEnum* function, from the *Enum* class.

The completion of missing (group index) values is solved by the *allvalues* function. This function can be used by a presentation function to complement the group of values, before rendering the presentation. By default, *allvalues* is set to *NoCompletion*, not requiring a presentation function to add missing values in a presentation. One would expect that the completion of missing values also requires a neutral value to be added for the missing value. But this is not strictly necessary, because most presentation rendering tools provide a special value for missing data, as we will see in a next section.

By choosing the *KeyHistory* data type as key the types for the presentation functions become:

$render1DTableSmart :: [(KeyHistory, value)] \to String$
$render2DTableSmart :: Ord\ a \Rightarrow [(KeyHistory, [(a, b)])] \to String$
$render2DTableSmart' :: [(KeyHistory, value)] \to String$

With this approach we now can render presentations, for which we can influence the ordering and the handling of missing values, by providing an instance for the *DataInfo* class for the data type of the group index. For the rendering of titles and the other components of a presentation, we can use a similar approach as presented in the previous section, now based on the information we encapsulate in the *KeyHistory* data type.

Current presentation implementations

The ideas from the previous section form the basis for the presentation functions implemented in our analysis library, which we discusses next. This and the following sections describe the practical details involved in using the functions, as well as the tools needed to render the actual plots. Currently, our library implements only presentation functions with the use of the *KeyHistory* key data type. The actual choice for a key type is normally not much of a concern for a user of our library, he just has to use the combinators and the presentation functions.

The library provides textual, as well as graphical presentations. The textual presentations are one-dimensional and two-dimensional tables. One-dimensional tables present data in a single column. A two-dimensional table presents analysis results based on (at least) two grouping analyses, displaying one grouping vertically, and the other horizontally. Provided graphical presentations are bar charts, stacked bar charts, and box plots. Also, relative stacked bar charts are available, for which each segment of the bar represent the fraction compared to the rest of bar.

The presentation functions have the following types:

$showAsTable1D :: Show\ b \Rightarrow [(KeyHistory, b)] \to MarkUpDoc$
$showAsTable2D :: (Show\ a, Ord\ a, Show\ b) \Rightarrow [(KeyHistory, [(a, b)])] \to MarkUpDoc$
$showAsTableDynamic :: Show\ b \Rightarrow [(KeyHistory, b)] \to MarkUpDoc$

$renderBarChart :: (Show\ b, Num\ b) \Rightarrow FilePath \to [(KeyHistory, b)] \to IO\ (FilePath, String)$
$renderStackedBarChart :: (Show\ a, Ord\ a, Show\ b, Num\ b) \Rightarrow FilePath \to [(KeyHistory, [(a, b)])] \to IO\ (FilePath, String)$
$renderRelativeStackedBarChart :: (Num\ b, Real\ b, Show\ b) \Rightarrow FilePath \to [(KeyHistory, b)] \to IO\ (FilePath, String)$
$renderBarChartDynamic :: (Show\ b, Num\ b) \Rightarrow FilePath \to [(KeyHistory, b)] \to IO\ (FilePath, String)$
$renderBoxPlot :: (Show\ b, Num\ b, Ord\ b) \Rightarrow FilePath \to [(KeyHistory, [b])] \to IO\ (FilePath, String)$

The presentation functions are very similar to the presentation functions described so far. A difference is the result types of the presentation functions. The result of the functions for rendering tables are of type

MarkUpDoc. From this data type we are able to generate LaTeX, a format which can easily be included in other LaTeX documents, as well as `HTML`, a format which can easily be browsed and published on line. The *MarkUpDoc* data type is in fact a wrapper around the *HTML* data type, from the *Text.Html* library. This library is a combinator framework for constructing and rendering `HTML` pages, and is part of the Haskell Hierarchical Libraries [6]. With a few extensions to this library, like a LaTeX rendering function, we created an easy solution for rendering `HTML` and LaTeX. We do realize that our implementation is a rather ad hoc solution, which might change in the future.

Note that the (stacked) bar chart and box plot rendering functions also require an extra parameter. This parameter, a file path, specifies where the image of the graphical presentation should be stored. The parameter does not require an exact filename. The presentation function is able to generate a filename based on the keys in analysis results. Often these keys describe the analysis relatively uniquely. Nonetheless, the user should be careful when generating large sets of analyses. In a future version we plan to provide safer solution.

Another difference is that the (stacked) bar chart and box plot rendering functions also return a tuple of the full path of the generated image file and a description. These are provided so that the image can be included in other documents, as we will see in examples in Section 5.2.7. The description of a presentation is returned, because the current implementation does not show this description in the image itself. The description can become very long, and would clutter the presentation. The presentation functions for (stacked) bar charts and box plots also return their results in the *IO* monad. This is because we use an external tool to render these graphical presentations.

The functions postfixed with '*Dynamic*' to their names, are presentation functions which ungroup the data, showing the last applied grouping in the extra dimension of the presentation. This means for a table presentation, that the last grouping will be shown in columns of a (two-dimensional) table. For a bar chart presentation this results in a stacked bar chart, which shows the last grouping in the stacked elements of each bar. This default behavior is sufficient for most cases. Variants for which the user can decide which grouping is shown, is scheduled as future work. We plan to also develop presentation functions for which the user can customize the rendering by setting custom presentation attributes (like custom titles, etc). Currently, none of the presentations functions show axis titles, something we also like to change in a future release.

Handling ordering and missing values

In the current version of our statistical library, we implement the ordering facility as described in the previous section, using the *DataInfo* class. The completion of missing values is only partly implemented. The presentation functions do not use the information from the *allvalues* function from the *DataInfo* class. However, the presentation functions do include missing group values, which we find, based on the input for two-dimensional presentations. The neutral value required to be added for a missing value, does not need to be specified, since our presentation rendering tool, `ploticus`, has a special sign ('`-`') for handling missing values. The same character is also used to specify missing values in table presentations.

However, when completion is required, the analysis developer has to take care for this. One can accomplished this manually by defining a special analysis which guarantees the presence of the required values. This suffices for most of the analyses in this thesis document.

Graphical presentation generation

The actual rendering of the graphical presentations, like bar charts and box plots, is done by a third party tool, called `ploticus` [4]. This tool is an actively maintained, free, GPL, non-interactive software package for producing a wide range of plots, charts, and graphics from data. It was developed in a Unix/C environment and runs on various Unix, Linux, and Windows systems. The presentation functions formulate ploticus script, which `ploticus` turns into graphical images. Supported graphical output formats are: `png`, `ps`, `eps`, `svg`, `svgz`, and others. Currently the plots are generated in `png` by default, which is suitable to be included in `HTML` as well as LaTeX documents.

Presentation interpretation

To conclude this section, we discuss the presentations available in the Neon library. We describe the table presentation, the different types of bar charts, and the box plot presentation.

In Section 5.2.7 two examples analyses are discussed, for which also two tables are shown. Figure 5.5 shows an example of an one-dimensional table, showing the number of compiles for each phase. In the rows we see the different phases. In the (second) column the total number of compiles is shown. Figure 5.8 shows a two-dimensional table, showing also the number of compiles per week. In this presentation the week numbers are shown in the vertical dimension, the rows of the table. The phases are shown in the columns of the table.

The analysis results shown in the tables can also be presented as bar charts. A simple bar chart, showing the data of Figure 5.5, is shown in Figure 5.4. The phases are positioned on the x-axis. The numeric values are shown as bars, each of a certain length, depending on the computed number of compiles. Similarly, Figure 5.7 shows the data of Figure 5.8. This bar chart shows the different weeks on the x-axis. The number of computed compiles per phases are represented by bar segments, stacked onto each other per week. Each bar tell something about the total number of compiles, as well as number of compiles per phase. However, it is not easy to see how the number of compiles are distributed per week. For this we can calculate the relative fractions, per phase. These values are presented in a relative stacked bar chart. An example of this presentation is shown in the phase analysis case study in Section 6.2.

A more complex presentation to interpret, is the box plot presentation (also known as a box-and-whisker diagram). In Figure 5.2 we show the number of compiles per day for each phase, in a box plot diagram. The phases are placed on the x-axis. The y-axis expresses the number of compiles. For each phase the presentation shows a box plot, representing a set of values, the number of compiles on each separate day of the course. Usually, an aggregate function is applied, to evaluate a set of values. For instance, the average, median, minimum, or an other aggregate metric is computed. A bar chart could be used to visually represent each of these aggregate values. However, we would lose the overview of the distribution of the separate values. To study this, a scatterplot could be used, but this does not tell us anything about any aggregate values. A box plot diagram, is a combination of both presentations, displaying a box and several line segments, together representing the complete data set.

The box of a box plot, represents the values between the lower and upper quartile, of the values from which a box plot is rendered. The lower quartile is the 25th percentile and the upper quartile is the 75th percentile, as such this box represents the middle 50% of the data. The line segments connected under and above the box plot, represent respectively the smaller and larger non-outlier observations. In the current implementation the line segments are extended the 5th and 95th percentile. Other implementations specify the outlier to be smaller or larger than 1.5 times the interquartile range, the difference between the third quartile and the first quartile. However, the current rendering provides us an easy way to formulate a conclusion about the extermes in the analysis results. Extreme values, smaller or larger than the 5th and 95th percentile, are shown as horizontal line segments, not connected to the box plot. The current implementation of the box plot presentation does not scale or leave out any value, so all values are somehow represented in either the box or a line segment. Furthermore, the median and the average of the values are displayed respectively as blue and green dots. The median is an aggregate value, which is less effected by extremes in the outcomes. The number under the x-axis shows the number of values, which are represented by each box plot, this number can differ per box plot.

5.2.7 Example combinator analyses

In this section we show the combined use of analysis combinators and presentation functions, introduced in the previous sections. First we discuss a rather simple example, computing the number of compiles for each phase. A second example, computes this also per week, which effectively adds a dimension to the computation and the presentation. This second example shows an approach which is also used for the case study analyses, discussed in Chapter 6. All presentations shown in this section are generated using Neon. No titles have been manually edited, only a minimum of changes are applied to improve the layout, like the alignment, of the images in this document.

Example 1: A minimal approach

A minimal implementation to execute and present the number of compiles per phase for a certain collection of loggings, is shown in Figure 5.3. To run this analysis, one can call the *presentLoggingPerPhase* from `GHCi` or integrate this function in the *main* function of a program.

The computational part of this analysis is constructed out of analyses, presented earlier this chapter, but this time we use the primitive combinators, available with the *DescriptiveKey* specialization. We also use

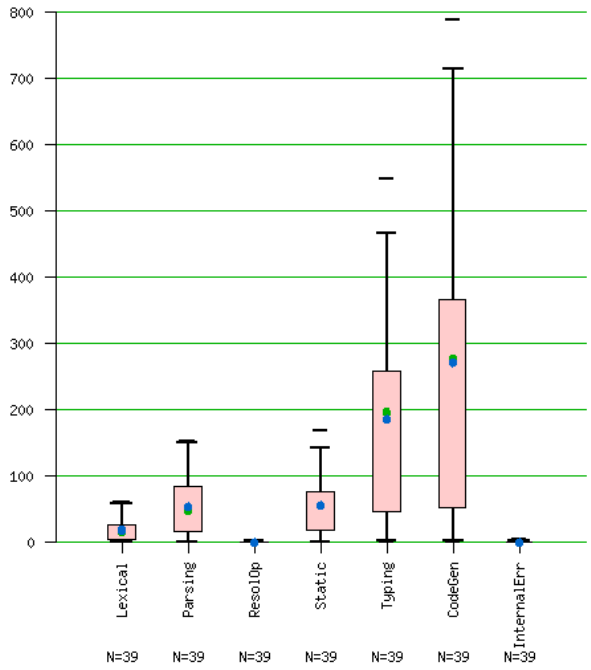


Figure 5.2: Box plot of number of compiles per phase per day

KeyHistory as key. So the type of *countNumberOfLoggings* can even be simplified a little more, similar as the type of *loggingsPerPhase*, namely: *AnalysisFK KeyHistory* [*Logging*] *Int*.

To execute this analysis one can call the *presentLoggingPerPhase* with respectively a path to a logfile and path were the output of this analysis can be stored. The presentation part of this analysis is an IO operation, because we use an external tool for rendering our graphical presentations. This is not a serious disadvantage since the handling of data, like writing it to a file, or presenting it on screen would turn out to be IO operation anyway.

The function *presentLoggingPerPhase* first parses the logfile using *parseLogfile* of type $FilePath \rightarrow IO\ [Logging]$. This parser is provided with this thesis work, and is discussed in Appendix E.1. The obtained loggings are then used to execute the analysis *loggingsPerPhasePerWeek*, using the *runAnalysis* combinator. From the result of the analysis, we generate a bar chart and a table, shown in Figure 5.4 and Figure 5.5. We need to explicitly render and store the table presentation in a file. The titles generated with the presentation functions, are as shown in the figures, as well as the labels on the x-axis of the bar chart.

The imports of this example show the modules and functionalities which are needed for this analysis. The library for statistical analysis and presentation is separated from the components used in our research. Each of the modules are shortly described in the Appendix C.

Example 2: A more general approach

The previous example is a minimal implementation of an analysis. When working with multiple analyses, it is beneficial to handle the common elements of such analyses, like parsing the logfile and executing the analysis, separately. One also would like to have the possibility to provide the preferred output format. These considerations lead us to a new concept, which also handles the presentations of the analysis results. For this we introduce a new type:

type $Research = OutputFormat \rightarrow FilePath \rightarrow [(KeyHistory, [Logging])] \rightarrow IO\ ()$

A function of type *Research* defines the analyses, as well as the presentations on the results of the analyses. For the rendering of the output, the first parameter can be used. The output is stored in the file path, provided as second parameter. The data is provided (including an initial analysis key) as third parameter. The result of this function is in the IO monad, because the function uses presentations, like also shown in the previous example.

```
module ThesisACExampleSimple where
import Statistics.Presentation
import Statistics.DescriptiveAnalysis
import Thesis.Helium.Logging
import Thesis.Helium.LogfileParser (parseLogfile)
import Grouping (groupAllUnder)

groupPerPhase :: DescriptiveKey key ⇒ AnalysisFK key [Logging] [Logging]
groupPerPhase = groupAnalysis phase (groupAllUnder phase)

countNumberOfLoggings :: DescriptiveKey key ⇒ AnalysisFK key [a] Int
countNumberOfLoggings = basicAnalysis' "number of loggings" length

loggingsPerPhase :: AnalysisFK KeyHistory [Logging] Int
loggingsPerPhase =
        countNumberOfLoggings
  <·> groupPerPhase

presentLoggingPerPhase :: FilePath → FilePath → IO ()
presentLoggingPerPhase logfile outputfp =
  do
    -- parse the logfile
    loggings ← parseLogfile logfile
    -- execute analysis
    let analysisResult = runAnalysis loggings loggingsPerPhase
    -- present analysis result
    barChart ← renderBarChart outputfp analysisResult
    writeFile (outputfp ++ "/analysis.tex")
              (renderLateX $ showAsTable1D analysisResult)
```

Figure 5.3: A minimal analysis, to compute the number of compiles per phase

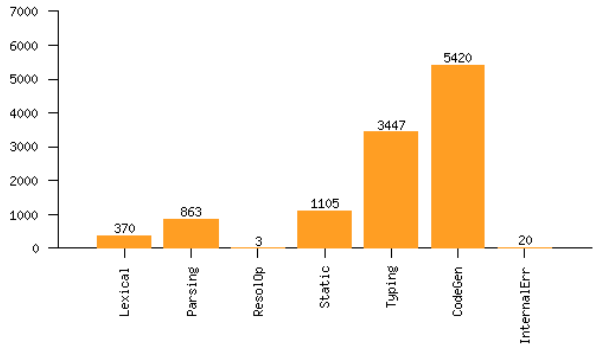


Figure 5.4: Bar chart of number of loggings per phase

	Number of loggings
Lexical	370
Parsing	863
ResolOp	3
Static	1105
Typing	3447
CodeGen	5420
InternalErr	20

Figure 5.5: Number of loggings per phase

```
module ThesisACExampleComplex where
import Statistics.Presentation
import Statistics.DescriptiveAnalysis
import Thesis.Helium.Logging
import Thesis.Helium.LogfileParser (parseLogfile)
import Thesis.Research.Common (countNumberOfLoggings, groupPerPhase,
                               groupPerWeek, groupPerStudent, plotToFigure, runResearch)
import Auxilary

loggingsPerPhasePerWeek :: AnalysisFK KeyHistory [Logging] Int
loggingsPerPhasePerWeek =
      countNumberOfLoggings
  <·> groupPerPhase
  <·> groupPerWeek
phaseResearch :: OutputFormat → FilePath → [(KeyHistory, [Logging])] → IO ()
phaseResearch format outputpath input =
  do
    barChartStacked ←
      renderBarChartDynamic outputpath (loggingsPerPhasePerWeek <$> input)
    barChartSplitPerStudent ←
      mapM (renderBarChartDynamic outputpath . loggingsPerPhasePerWeek)
           (splitAnalysis $ groupPerStudent <$> input)
    -- show result data in tables
    writeFile (outputpath ++ "/example" ++ "." ++ (showFormatExt format))
              $ renderMarkUpDoc format $
                showAsTableDynamic (loggingsPerPhasePerWeek <$> input)
                  +++
                plotToFigure barChartStacked
                  +++
                concatHtml
                  (map plotToFigure barChartSplitPerStudent)
```

Figure 5.6: A more complex analysis, to compute the number of compiles per phase, per week

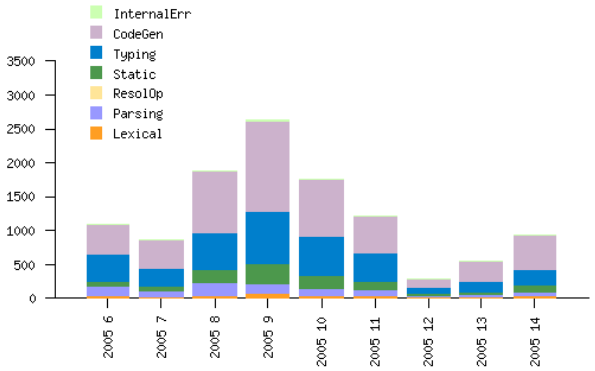


Figure 5.7: Bar chart of number of loggings per phase per week

	Lexical	Parsing	ResolOp	Static	Typing	CodeGen	InternalErr
2005 6	49	132	-	81	392	438	1
2005 7	29	88	-	67	258	416	-
2005 8	50	182	-	192	552	910	1
2005 9	78	148	-	293	766	1336	17
2005 10	42	101	3	187	587	835	1
2005 11	45	85	-	122	418	546	-
2005 12	18	29	-	30	88	126	-
2005 13	18	35	-	37	163	293	-
2005 14	41	63	-	96	223	520	-

Figure 5.8: Number of loggings per phase per week

The evaluation of such a function is handled by *runResearch* :: *Research* → *KeyHistory* → *FilePath* → *FilePath* → *IO* (). This is a helper function, which runs the research analysis with a startkey, a logfile and a proper output directory. This function takes care for the common task or reading and parsing the logfile. Basically it is not much different then the previous example, but now we separate the handling of the in- and output, from the actual analysis and analysis presentation work.

An example of an research analysis is shown in Figure 5.6. Here we analyze the number of loggings per phase per week. This is calculated by the *loggingsPerPhasePerWeek* analysis. From the analysis result of this analysis we render a table and a stacked bar chart in the *phaseResearch* function as shown in Figure 5.8 and Figure 5.7. The analysis can also be used to analyze this subject per student, as shown in the example as well. The result of the analysis is put together in a single document, by using *MarkUpDoc* combinators. The values allowed for the output format of type *OutputFormat* are *LaTeXFormat* and *HtmlFormat*.

To run this example the *phaseResearch* can be called from GHCi, using the *runResearch* function. The research can also be integrated in the *main* of a program. For this thesis we developed a special wrapper tool, as part of the analysis tool set. This tool is able to execute the analysis (in the form of a research) and takes care for practical issues, like checking whether required paths and files exist. Furthermore, it can also run a set of analyses. This tool is discusses in Section 5.3.3.

5.3 Tool set utilities

The tool set of our research consists of several components, from which NEON, the descriptive statistics library, forms the most prominent part. In addition, also a set of shell and Perl scripts were developed. These utilities support the process of collecting and analyzing Helium loggings, and will be discussed in this section.

Section 5.3.1 discusses the utilities which help in handling multiple Helium compilers, without the need to explicitly set environment variables. In Section 5.3.2 we discuss scripts and tools that support the preprocessing of a logging data set. Section 5.3.3 discusses a tool which provides a command line interface for analyses designed using the NEON library, discussed in Section 5.2.

5.3.1 Calling Helium and Helium components

The Helium compiler and its related tools, like `hint`, `texthint` and `lvmrun`, require that the environment variable `LVMPATH` is set correctly. Working with multiple compiler versions, requires the user to set or reset this variable, each time a different compiler is used. The following utilities take care of calling Helium or one of the related Helium tools with the appropriate setting.

```
helium.pl
helium.sh
hint.sh
texthint.sh
lvmrun.sh
```

The scripts can be used according to the following syntax:

```
helium.pl -sv <helium version> <helium options>
```

The value for the `sv` parameter refers to the new version number as described in Section 3.2.2 in Chapter 3. Calling the script without the `sv` parameters, selects the Helium 1.2 compiler by default. Using these utilities, requires to set the variable `$heliumbinpath`, inside each script. This variable must refer to the directory of the installed Helium compilers. This variable only needs to be changed, if the location of the Helium compilers is changed.

The `sethelpaths.sh` script can be used to only set the required path for a specific Helium compiler.

Syntax:

```
source sethelpaths.sh [-sv <version, eg.: 1.2, 1.5>]
```

The `source` command runs the script in the current shell environment, and therefore also effecting the environment variables of the current shell. Calling the script without the `-sv` parameters, selects the Helium 1.2 compiler by default.

5.3.2 Helium logging preparation scripts

Working with sets of Helium loggings is supported by various utilities. These utilities are discussed in this section.

Logging data set compilation

The `heliumlogcompilation.pl` utility can be used to reproduce the compiler output of a complete set of loggings. Several parameters are available to specify which compiler should be used, and which optional preprocessing steps should be applied to the source.

Syntax:

```
heliumcompilation <logfile> <version> <script> <logoutput> [<compiler options>]
```

The first parameter specifies which logfile to use. A logfile is placed in the parent directory of a collection of Helium loggings, and describes the complete set of loggings. Lines starting with a `#` sign are not compiled by this utility.

With the `version` parameter, one can specify which compiler version should be used to compile the loggings. Allowed values are the new Helium version numbers, as described in Section 3.2.2 in Chapter 3 (for instance: `1.1`). Allowed values are also the combination of a version number with a build date of a compiler (for instance: `1.1-Mon-Apr-7-16_57_57-2003`). Other allowed values are `current`, to use the compiler available in the search path, or `from-log`, to use the compiler version mentioned in logfile.

With the `<script>` parameter, one can specify a script, which will be run before the compilation takes place. This parameter can be used to fix any incompatibilities in the source in relation to the used Helium compiler. For instance, since Helium 1.5, the compiler changed the way it handles exports in the module declaration. As a consequence Helium 1.5, or any newer versions, are not able to always compile loggings developed with older Helium compiles. The specified script receives as first parameter the full path and filename of the logging, and as second parameter the logfile line of the logging. Providing the string `"none"`, disables this option.

With the `<logoutput>` one can specify how the output should be formatted. Allowed values are: `full` and `simple`. `full` renders similar information for each compilation like the line specifying the logging in the logfile. The only difference is the absence of the static error codes. `simple` provides only basic information on each logging compilation. `simple` runs slightly faster, especially on bigger logging sets, because it does not analyze the compiler output for information, like the resulted phase.

Resolving module declaration change

Since Helium 1.5 the compiler does not allow exports to be specified in the module declaration. To be able to still compile older loggings, for which this was allowed, `fixHel15Issues.pl` can be used.

Syntax:

```
fixHel15Issues.pl <fullpath> <logfileline>
```

This scripts edits all files found in the `fullpath` parameter and applies a regular expression to it, removing all module exports. This script can be used as parameter to `heliumlogcompilation.pl`.

Locating import scripts

As from Helium 1.5 (released January 2005) the logger also sends the import modules along with each logging. For data sets logged before January 2005, one had to locate the imports of a logging, to be able to recompile the logging. The import can often be found in the earlier loggings of the programmer. This process of finding imports is described in Section 3.2.1. The process of locating imports is partially automated by the following scripts.

The utility `getallimports` finds all imports in a set of loggings.

Syntax:

```
getallimports <filename>
```

This script looks for source files (*.hs) in the subdirectories of the current directory. If these source files have imports, then their name and the name(s) of the module import(s), are outputted to a file. The filename of this file must be specified as parameter.

The utilities `globalFixImport.pl` and `fiximport.pl` are able to locate import modules in a set loggings and produce a set of script rules, which copy the import modules to the required logging.

Syntax:

```
globalFixImport.pl <import module>
fiximport.pl <import module>
```

The scripts work as follows, first the `globalFixImport.pl` is called with a particular (import) module name. This script analyzes the directory structure of the loggings, which are grouped per user name (each having a directory). For each directory, the script `fiximport.pl` is called. The `fiximport.pl` script keeps track of the specified import module in a set of loggings. If the script runs into a logging, which imports the module, it can precisely tell from where the import should be copied. The script `fiximport.pl` outputs for each found import a rule, specifying how to copy the import (if available), as a simple `cp` command. The resulting output, forms a script which can be run to correct the missing imports. The script `fiximport.pl` also displays the difference in time between the module and the source file. Furthermore, the script warns for cases where no import could be found, and for cases where the import has imports of its own.

Logfile updating scripts

Until Helium 1.5, the logger did not store the name of the program being logged in the logfile. There was no direct need, since there was only one file being logged. After locating and copying module imports to the logging, it is necessary to have the filename of the main program specified in the logfile. The utility `addMainFileToLog.pl` is a script which adds the main file to each logging mentioned in the logfile. Note that this script should be applied before correcting any missing import modules in the set of loggings.

Syntax:

```
/users/user/heliumdata] analyze -X
Helium Logging Analysis 1.0 Build date: Sun Aug 13 20:31:35 UTC 2006
Usage: analyze [options] logfile
  -a 0           --analysis-reference=0      run predefined analysis over input
  -A             --all-analyses              run all predefined analyses
  -X             --show-predefined-analyses  show description for all predefined analyses
  -o /tmp/       --output-path=/tmp/         output path for analysis results
  -m HtmlFormat  --output-format=HtmlFormat  output format, LateXFormat or HtmlFormat
  -h             --help                      shows this information
  -a 1                                       Research 1: Module length / Lines of Code
  -a 2                                       Research 2: Time and effort to repair a type (..)
  -a 3                                       ...
```

Figure 5.9: Analysis tool options

```
addMainFileToLog.pl <logfile>
```

The loggings are stored in a directory structure. The time stamp is part of the path referring to the logged program. This time stamp uses a colon (':') to separate the time elements. From a practical point of view one would like to replace them by a more suitable character. The script `rename.pl` replaces the colon with an underscore ('_').

Syntax:

```
rename.pl
```

The logging server, written in Java, suffered from a bug in the date function, causing the month to be one off. The utility `addMonthToLogfile.pl` can be used to correct this. Applying this script, avoids problems in parsing the dates.

Syntax

```
addMonthToLogfile.pl <original logfile> <newoutputpath>  [ > newlogfile ]
```

The first parameter specifies the logfile, which has to be corrected. The second parameter specifies the path to which the loggings are to be copied, correcting also the directory names. The output of this script are the loglines with the corrected month, which can be redirected to a new logfile.

The output from the `heliumlogcompilation.pl` script is similar to the content of a logfile, but without the static error codes. To be able to compare the original logfile with the reproduced logfile, using a command like `diff`, one has to strip the static error codes from original logfile. The utility `removeStaticErrorsFromLogfile.sh` can be used for this.

Syntax:

```
removeStaticErrorsFromLogfile.sh <logfile>
```

5.3.3 Calling combinator analyses from the command line

Analyses constructed using Neon, the statistical combinator library, can be run using GHCi, the interpreter of GHC). To achieve a better performance for the analyses of our research study, an analysis tool was created which executes predefined analyses from the command line. This section gives a short overview how the tool can be used and how it can be extended to include more analyses.

The wrapper tool is simply called `analyze` and has the syntax:

```
analyze [options] logfile
```

The options are shown in Figure 5.9.

A typical command to execute an analysis is:

```
> analyze -a 1 -o /data/outcome/fp0304/   \
>   -m HtmlFormat /data/loggings/fp-0304-p3/fp0304.log
```

This command runs an analysis, studying aspects related to module lengths, for loggings specified in the logfile `/data/loggings/fp-0304-p3/fp0304.log`. Output is written in `HTML`, to the `/data/outcome/fp0304/` directory.

The analysis utility provides a command line interface to the analyses developed for our research. To extent this set of predefined analyses, one should write an analysis similar to the example in Section 5.2.7, based on the discussed *Research* type, and add this analysis to *predefAnalyses*, a list of predefined analyses, defined in the module *Thesis.AnalysisPredefined*.

predefAnalyses = [
("`Module length / Lines of Code`", *moduleLengthResearch*)
,("`Time and effort to repair a type incorrect program`", *typingTillCodeGenResearch*)
, ...
]

Chapter 6

Analysis case studies

This chapter discusses analyses applied to Helium logging data sets collected during three incarnations of a functional programming course at Universiteit Utrecht. Each analysis is an example study, displaying the use of the combinator library, discussed in Chapter 5. With each study we provide a first view on the studied topic. We start by discussing a motivation for doing the analysis. From this we derive a claim for which we design a concrete analysis. The analysis is implemented using the statistical combinators, specialized by the *DescriptiveKey* class and using the *KeyHistory* data type as key implementation. For each analysis we present the outcomes, for a certain logging data set. Based on these outcomes, we discuss the validity of the claim. Because we use only descriptive statistics, we do not imply or infer anything beyond the studied sample. In addition, we also discuss shortly further possible research related to the topic of study. It is a future goal to evaluate the Helium compiler more systematically.

The analyses presented in this chapter range from relative simple analyses to more complex ones. Section 6.1 studies the size of logged programs, analyzing the module length per week and per weekday. In Section 6.2 we compute how many logged programs end in each compilation phase. In Section 6.3 we analyze the content of logged programs and look at how many lines are code, comments, or empty lines. Section 6.4 calculates the time or interval between compilations. In Section 6.5 we analyze the time needed to repair a type incorrect program. Section 6.6 studies the level of hints in type error messages. The analyses from Section 6.5 and Section 6.6 are combined in Section 6.7, where we study the effect of hints on the time needed to repair a type incorrect program.

6.1 Module length analysis

A metric often used in software quality management is the lines of code (LOC) metric. This metric includes code lines, comments and empty lines. In large software projects the (total) size of a software system is considered a good prediction of the quality and the likelihood for errors. In functional programming a program is often more compact, compared to imperative or object oriented programs. A functional programmer can keep programs concise by using higher-order functions and other advanced abstraction mechanisms. Nonetheless, for novice programmers, with less skills, an increase in size of a program is a good indicator that additional functionality was added.

In this analysis we study the length of logged source modules in number of lines of code. We expect this number to be growing over time. Each of incarnation of the functional programming course also involved graded assignments, as described in Section 3.1.1. Consequently, we expect that the average module length peaks around the deadline of assignments. After a deadline and with the introduction of a new assignment, the average module size drops in value and again will increase over time. From this we formulate the following claim:

Claim: Module sizes increase over time, towards the deadline of an assignment. This pattern repeats itself for each assignment.

Analysis design

To analyze our claim we calculate the average module length for a given set of loggings per week and per day. The implementation of this analysis is relatively easy. We first define an analysis which calculates the module length of a logging:

```
locMetric :: AnalysisFK KeyHistory Logging Int
locMetric = basicAnalysis "module length (LOC)" (length . lines . sourcecode)
```

We call this analysis a metric, because it calculates a value from a single, independent logging. The source of a logged module is provided by the function *sourcecode* of type *Logging* → *String*. The implementation of the *Logging* data type and the *sourcecode* function are discussed in Appendix E.2.

Next, we construct the analyses *metricAnalysisPerWeek* and *metricAnalysisPerWeekDay*. These analyses group a set of loggings per week and per day of the week, and subsequently map a metric analysis over the resulting sets of loggings. For the current analyses we use the metric *locMetric*. This allows us to reuse these analyses with other metrics.

```
type Metric a = AnalysisFK KeyHistory Logging a
metricAnalysisPerWeek :: Metric a → AnalysisFK KeyHistory [Logging] [a]
metricAnalysisPerWeek metric =      mapAnalysis metric
                              <·> groupPerWeek

groupPerWeek  :: AnalysisFK KeyHistory [Logging] [Logging]
groupPerWeek  = groupAnalysis (weekOfYear . logDate) groupPerWeek_
groupPerWeek_ :: [Logging] → [[Logging]]
groupPerWeek_ = ...

metricAnalysisPerWeekDay :: Metric a → AnalysisFK KeyHistory [Logging] [a]
metricAnalysisPerWeekDay metric =      mapAnalysis metric
                                 <·> groupPerWeekDay
                                 <·> groupPerWeek

groupPerWeekDay :: AnalysisFK KeyHistory [Logging] [Logging]
groupPerWeekDay = groupAnalysis ctWDay (λl1 l2 → compare (ctWDay l1) (ctWDay l2))
```

The analysis *metricAnalysisPerWeekDay* uses *groupPerWeekDay* in combination with *groupPerWeek*. The analysis *groupPerWeekDay* groups loggings per day of the week (so all loggings from Mondays are grouped together and so on). This approach is taken for its ease of implementation, since *ctWDay* :: *CalendarTime* → *Day* is a field label of the *CalendarTime* data type, provided by `System.Time` library as part of the Haskell Hierarchical Libraries [6].

To aggregate over the calculated module lengths we define:

```
averageAnalysis :: (DescriptiveKey key, Real a) ⇒ AnalysisFK key [a] (OptData Float)
averageAnalysis = basicAnalysis "average" mean

average :: Real a ⇒ [a] → OptData Float
average []   = NaN
average list = Value $ realToFrac (sum list) / realToFrac (length list)
```

Similarly to *averageAnalysis*, we also provide analyses to calculate the median, minimum, maximum, etc. The result of *averageAnalysis* is, as well as other aggregation functions, of type *OptData Float*. The *OptData* data type is similar to *Maybe*, and is defined as:

```
data OptData a = Value a
               | NaN
```

We use this data type, because it allows us to handle empty groups, returned by an analysis. In some cases one would like to maintain the empty groups because this is relevant for later calculations. Consequently, a calculation over such a set of values must also be able to handle empty lists. This custom data type also allows us to conveniently define a custom *Show* implementation. This eases the presentation functions for this data type. We also could have designed a custom type class for this.

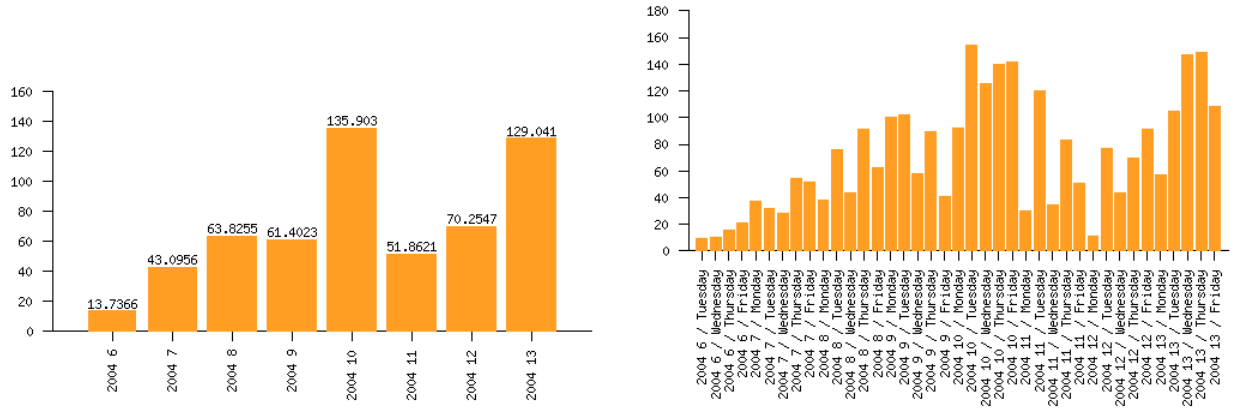


Figure 6.1: Average module length in terms of lines of sourcefile per week (left) and per day (right) based on the 2003/2004 data set.

From these analyses we construct an analysis which calculates the average module length per week:

$$moduleLengthPerWeek = averageAnalysis <\cdot> metricAnalysisPerWeek\ locMetric$$

Similarly we also construct an analysis which calculates the module length per day of the week, by using *metricAnalysisPerWeekDay*. Results from these analyses can be presented with presentation functions, described in Section 5.2.7.

Analysis results

To support our claim we provide bar charts, shown in Figure 6.1, presenting the average module length per week and per day from the 2003/2004 programming course, for the complete data set. In this course, students had to use Helium for the first two assignments. The deadline for the first assignment was March 14, 2004 (Friday of week ten). For the second assignment this is March 26, 2004 (Friday of week thirteen). For the third assignment students had to use `Hugs` instead of Helium, therefore no loggings were made in the weeks 14, 15 and 16.

The average module size per week, shown in the left-hand side of Figure 6.1, shows an increase of the average module size per week, for each assignment. The weeks in which the assignments were to be handed in, show a value of almost twice as much as for the week before the deadline. When we study the average module size per day, shown in the right-hand side of Figure 6.1, we see a similar development. Although the values do increase, this is not monotonous. Some days also show very low reading, like for instance Friday of week 9.

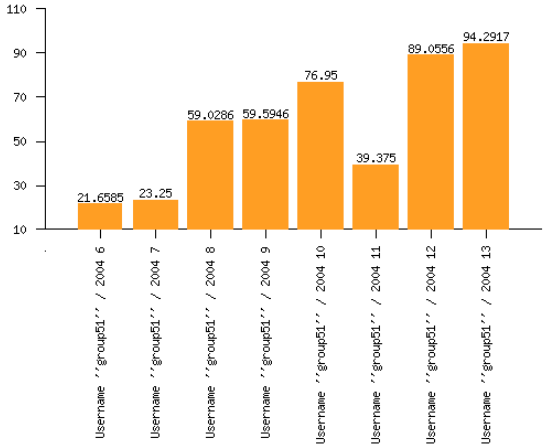
Furthermore, we also analyzed this aspect per student. For practical reason we provide the graphs of only two students in Figure 6.2. For clarity the plots displaying the module length per day (on the right-hand side), only show the days on which the student actually worked and loggings were made.

Both students are very active and show an increase in lines of code, with peaks around the deadlines, like we saw for the complete data set. Student 51 appears to be writing smaller programs compared to the rest of the students. We also see that the graphs show similar shapes for both students, compared to the rest of the data set.

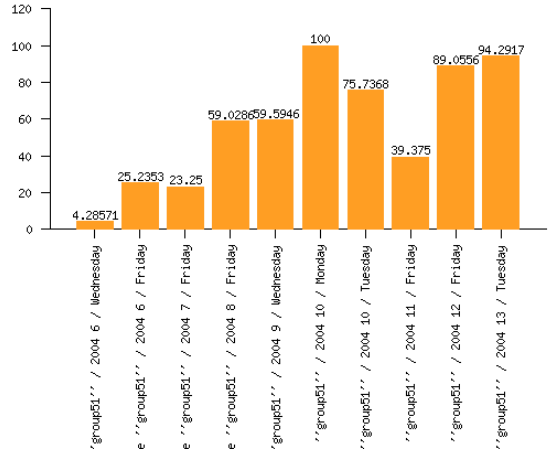
Conclusion and further analysis

From the statistics shown in the previous section we assume that the claim of increasing module lengths holds. This also holds for the two students, analyzed separately from the rest. Further analysis needs to be done to show the value of this analysis. For instance, one could study the relation between the number of lines of code and added functionality to a program.

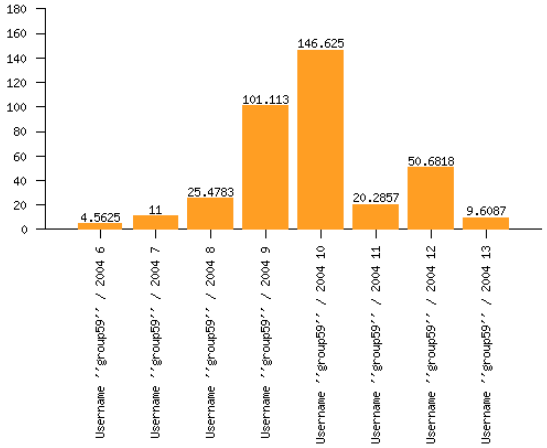
Future research could extend this analysis to study also aspects related to the graded program. The filename and content of a logged program often indicates on which assignment the student is working. Combining this



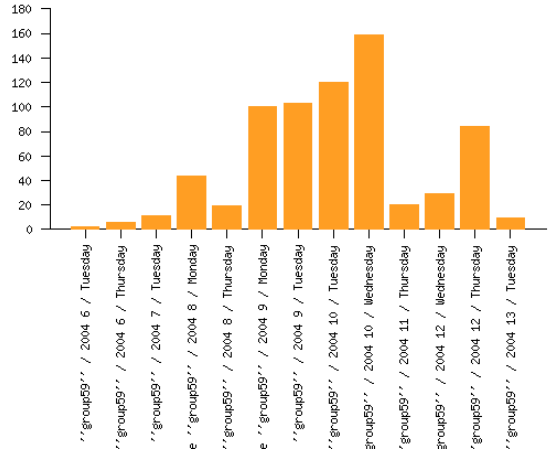
(a) Student 51



(b) Student 51



(c) Student 59



(d) Student 59

Figure 6.2: Average module length per week (left) and per day (right) for two particular students from 2003/2004

information with the presented analysis, can analyze the influence of an early or late start on the grade or the quality of the resulted program. Also could be studied whether the result of this analysis per student, in relation to other students, is a good predictor for the progress in the development of a graded assignment. In a future application, such a predictor could warn the student, teacher, or lab assistant, when the student falls behind.

Another direction of research is to analyze the actual content of the programs. For instance, one could study the ratio of code, documentation, and empty lines of a program. This analysis is presented in Section 6.3.

6.2 Phase analysis

Helium is designed to be used in an educational setting. The high-quality error messages are generated by using advanced compiler construction techniques. The compiler is kept maintainable by having a sequential staging of the compilation process. Each compilation step is fully separated and handles a specific range of errors. While programming, a student writes a program which requires to pass the phases of lexical analysis, parsing, resolving operators, static analysis and typing inferencing, before an executable is returned. In time and through practice, a student is expected to get experienced in getting a program to the next phase. This yields the following claim:

Claim: Over the course of time, the ratio of successful compiles increases.

Analysis design

To find support for our claim, we design an analysis to calculate the number of loggings per phase, per week. The outcome of such an analysis can be presented in a table and a stacked bar chart. These absolute numbers are useful to ensure that we involve enough loggings in our analysis per phase. As such, we can study the growth in number for each phase. This does not tell us much about the relative performance per phase over time. To study this, we also calculate the fractions (or the percentages) of the number of loggings per phase, compared to the total number of loggings per time unit. We also present these results in a table and a stacked bar chart.

If our intuition is correct, we will see an increase for the fraction of loggings which compile successfully, the loggings which end in the code generation phase. Consequently, we expect a decrease for some or maybe all phases which generate error messages. The phases in which errors can be detected are the lexical phase, the parsing phase, static analysis phase, resolving operators phase, and the typing inferencing phase. We leave out the phase of resolving operators, because this phase rarely causes an error.

We design our analysis slightly different from the module length analysis. We first define an analysis to group loggings per phase, using the *groupAnalysis* primitive.

```
groupPerPhase :: DescriptiveKey key ⇒ AnalysisFK key [Logging] [Logging]
groupPerPhase = groupAnalysis phase (groupAllUnder phase)
groupAllUnder :: Eq b ⇒ (a → b) → [a] → [[a]]
groupAllUnder = ...
```

The function *phase* is of type *Logging* → *Phase* and is a field label of the *Logging* data type. The *Logging* data type, the provided field labels, and additional helper functions are discussed in Appendix E.2.

With the *groupPerPhase* combinator we construct the *loggingsPerPhase* analysis, which groups and counts the loggings per phase.

```
loggingsPerPhase ::      AnalysisFK KeyHistory [Logging] Int
loggingsPerPhase =           countNumberOfLoggings
                          <·> groupPerPhase
                          <·> mainPhasesAnalysis
mainPhasesAnalysis :: AnalysisFK KeyHistory [Logging] [Logging]
mainPhasesAnalysis = basicAnalysis "" (filter (anyfrom mainphases . phase))
  where
    mainphases =  [Lexical, Parsing, Static, Typing, CodeGen]
```

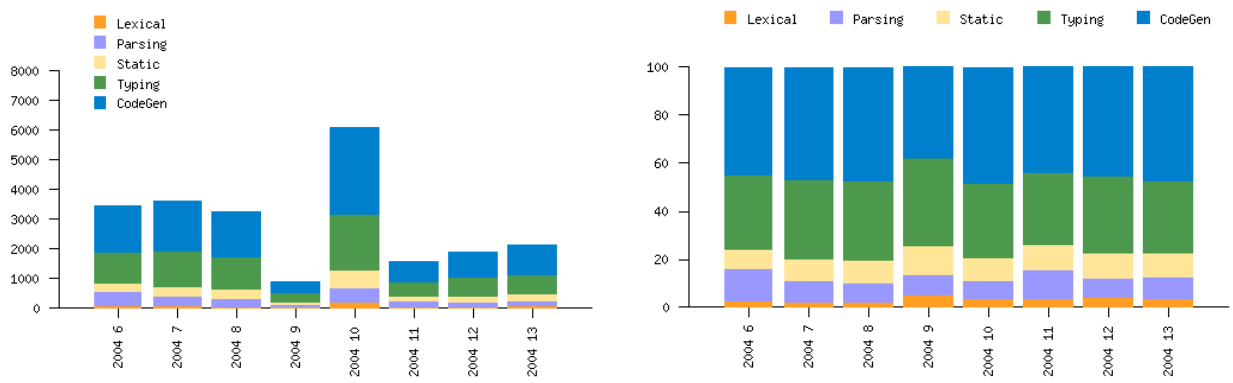


Figure 6.3: Absolute and relative number of compiles per phase, given per week from 2003/2004.

```
    anyfrom     =  flip elem
countNumberOfLoggings :: DescriptiveKey key ⇒ AnalysisFK key [a] Int
countNumberOfLoggings =    basicAnalysis "number of loggings" length
```

The analysis *loggingsPerPhase* first selects the phases in which we are interested, using the *mainPhasesAnalysis* analysis. The *mainPhasesAnalysis* analysis excludes the phase of resolving operators. This phase rarely causes an error and is therefore ignored. The internal errors are excluded; they are strictly speaking also not part of any specific compilation phase. The analysis *loggingsPerPhase* can be used in combination with the analysis *groupPerWeek*, presented in Section 6.1. The presentation functions *renderBarChartDynamic*, *renderRelativeStackedBarChart*, and *showAsTableDynamic* can be used to generate presentations of the absolute and relative numbers per phase.

Analysis results

To study our claim we present the absolute and relative number of compiles, from the 2003/2004 functional programming course, per phase, per week in stacked bar charts in Figure 6.3 and the actual values in Figure 6.4.

The absolute number of compiles per phase, shown on the left-hand side of Figure 6.3, show that the numbers strongly vary per week. The graph shows a low reading for week 9, which is in fact a holiday week. For this reason we ignore the results from this week (also in later analyses). A probable explanation for the high number of compiles in week 10 is because of the deadline of a graded assignment that week. All later weeks show a relatively low (but increasing) number of compiles. The reason for this remains unclear. It is likely a combination of more students working together, an increase in the number of students working with Helium on their computer at home, and a number of students giving up on the course.

The relative number of compiles per phase, shown on the right-hand side of Figure 6.3, provides a view on the effect of increasing experience on the fraction of failed compiles per phase. We expected to see an increase in the number of correct compiles, the code generation phase. However, we do not see a clear and overall effect. More detailed study of the actual values presented in Figure 6.4 reveals that the number of correct compiles seem to increase up to week 10, when ignoring week 9. The increase is mostly due to a decrease in the number of parse incorrect programs. In week 11, when students probably start working on a new (graded) assignment, the fraction of correct compiles seems to ”reset”, similar to week 6.

Similar analyses can be performed for each single student. We leave these studies for future research.

Conclusion and further analysis

To conclude this analysis we can say, that we do not see an overall increase in the number of correct compiles for the complete duration of the functional programming course. However, we do see a decrease for the period in which an assignment had to be developed. This provides us the evidence that the claim does hold for these two separate periods. The increase is mostly the result of a decreasing fraction of parse incorrect compiles, but whether this is significant enough remains to be investigated.

	Lexical	Parsing	Static	Typing	CodeGen
2004 6	90 (2.6%)	481 (13.8%)	276 (7.9%)	1073 (30.9%)	1553 (44.7%)
2004 7	92 (2.5%)	308 (8.5%)	339 (9.4%)	1195 (33.0%)	1684 (46.5%)
2004 8	76 (2.3%)	258 (7.9%)	314 (9.6%)	1074 (33.0%)	1536 (47.1%)
2004 9	48 (5.4%)	75 (8.4%)	105 (11.8%)	326 (36.6%)	336 (37.8%)
2004 10	227 (3.7%)	471 (7.7%)	585 (9.6%)	1890 (30.9%)	2934 (48.0%)
2004 11	56 (3.6%)	192 (12.3%)	159 (10.2%)	474 (30.3%)	685 (43.7%)
2004 12	78 (4.1%)	157 (8.2%)	194 (10.2%)	612 (32.1%)	867 (45.4%)
2004 13	81 (3.8%)	187 (8.7%)	223 (10.4%)	641 (30.0%)	1007 (47.1%)

Figure 6.4: Absolute number and percentage of compiles per phase, given per week from 2003/2004.

Another future research direction is the analysis of actual errors in a particular phase. Analyzing the source of an error could reveal information whether the error is indeed pointing to the correct error location and whether the error is detected in the correct phase in the first place. This kind of analyses are not easy to automate and would require manual inspection of the program source, which is a time consuming job. Also, could be studied when certain language constructs are introduced during the functional programming course. These new constructs often introduce several new parsing and typing errors.

Also, the educational background of the student can be of interest. The most common studies of the attending students functional programming are computer science, information science, and cognitive artificial intelligence. Some students might have prior experience in functional, imperative, and object oriented languages. These students are likely to make different errors based on their experience. Analyzing the subjects from each group separately can reveal the effect of prior knowledge.

6.3 Module segmentation analysis

When looking at the statistics on module lengths, one can wonder how many lines are actually lines of code, documentation, or just empty. Students are stimulated to write documentation for the graded assignments in the comments of the source. Having a properly documented assignment is rewarded with a better grade. The idea is that effective documentation helps the student during the process of programming and increases the understanding of a program. This yields the following claim:

Claim: Students document their programs throughout the development process.

Analysis design

To find statistical support for this claim, one has to analyze the source of each logging by splitting the source into lines (or segments) of code, comments, and empty lines. We assume that comments are mainly used to document a program, providing us a metric for the level of documentation of a program. Correctly analyzing the source is not as easy as it may sound. There are several details to take care of and it is essential to correctly detect the start and ending of comments. Helium supports single line and (multi line) nested comments. Nested comments may appear in the middle of code segments and are also allowed inside nested comments. Furthermore, we regard lines filled with spaces and tab characters as empty lines.

As a solution we reuse the lexer of the Helium compiler. The lexer splits the source in lexical tokens. This process detects and throws out all comments. Based on this lexer, we implement our own version, which analyzes the source according to the following rules:

- *A single line comment counts as one line of documentation*
- *A (multi line) nested comment counts as documentation for its length in lines. This also holds for nested comments inside nested comments, which are counted separately.*
- *Code lines are lines of text, other than lines of comments, which hold characters other than only whitespace characters.*
- *A code line containing a single line comment is counted as one code line*

- *If nested comments are present in a code line, the code line is considered to be split, resulting in two or more code lines. For example, if a nested comment is placed in the middle of a line of code (e.g.* `"display str {- string to display -} = putStrLn str"`*), the line is counted as two code lines and one comment line.*
- *All other lines, which from start to end consist out of whitespace or no characters at all (but with a trailing newline), are considered to be empty lines. (For anonymization reasons, the contents of comments were replaced by empty lines and the original empty lines in the comments are therefore not recognizable as such.)*

These rules describe the mechanism which categorizes the lines of a program file. Inevitably, some lines of text are counted twice, since we do want to count a line of code with comments in two ways. Based on the fifth rule, code lines are sometimes also counted more than twice. The latter is mainly because it allows an easy implementation of the custom lexer.

This lexer is implemented by the function *segmentlexer* of type $String \rightarrow [CodeSegment]$. A source segment is either one or more lines of code, comments or empty lines. The data type *CodeSegment* is defined as:

```
data SourceSegment =
    CodeLine String
  | SingleCommentLine String
  | MultiCommentLine String
  | EmptyLine String
  deriving (Show, Eq, Ord)
```

This data type represents the different types of source segments. The encapsulated string of each segment type holds the line(s) from the source. The result from the *segmentlexer* can be flattened for easy analysis by the *flattenSegments* function of type $SourceSegment \rightarrow [SegmentType]$, which evaluates the source string into a list of segment types. The *SegmentType* data type is defined as:

```
data SegmentType = CodeLn | CommentLn | EmptyLn
  deriving (Show, Eq, Ord)
```

With these functions we construct *lineSegmentMetric*, a metric analysis to calculate the source segmentation.

```
lineSegmentMetric :: AnalysisFK KeyHistory Logging [SegmentType]
lineSegmentMetric =
  basicAnalysis "Line of Source" (concatMap flattenSegments . segmentlexer . sourcecode)
```

The function *sourcecode* is a helper function for handling loggings and is discussed in Appendix E.2. With this *lineSegmentMetric* we construct an analysis which calculates the number of lines of code, comment, and empty lines per logging.

```
countSourceSegmentsMetric :: AnalysisFK KeyHistory Logging [(SegmentType, Int)]
countSourceSegmentsMetric =
      basicAnalysis "" countOrdValues
  <·> lineSegmentMetric
countOrdValues :: Ord a ⇒ [a] → [(a, Int)]
countOrdValues = ...
```

We use this analysis to construct *segmentDistribution*, an analysis to calculate the number of segments for a set of loggings.

```
segmentDistribution :: ([Int] → b) → AnalysisFK KeyHistory [Logging] [(SegmentType, b)]
segmentDistribution aggregatefun =
      basicAnalysis "" (map aggregateList . (groupAllUnder fst) . concat)
  <·> mapAnalysis countSourceSegmentsMetric
where
  aggregateList list = (fst $ head list, aggregatefun $ map snd list)
medianSegmentDistribution = segmentDistribution median
```

The *segmentDistribution* analysis can be parametrized by an aggregation function to abstract over the calculated numbers of lines of each segmentation. In this research we use the median, since this is considered to be a better measure to describe the central tendency of calculated values.

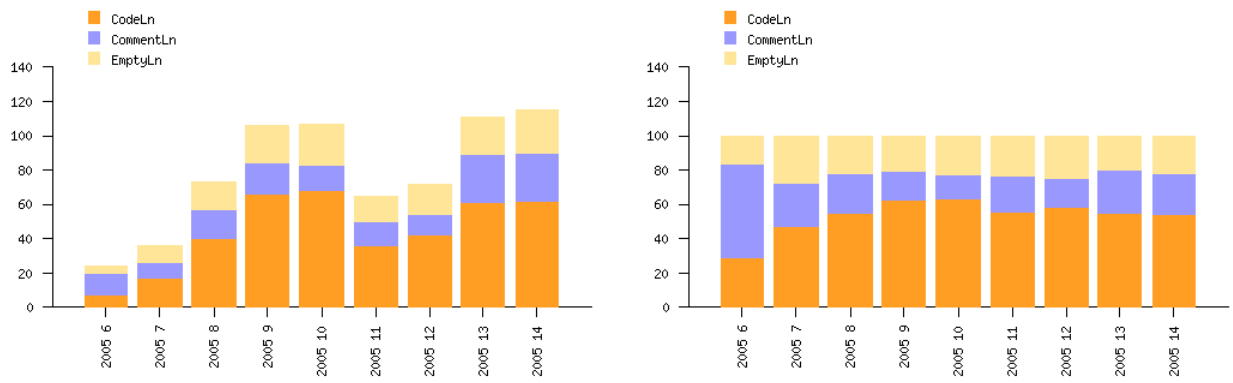


Figure 6.5: Median size (left) and relative fractions (right) of source program segments for all students, given per week from 2004/2005.

To present the analysis results of *segmentDistribution*, we now have to use an approach slightly different from the previous analyses. The *segmentDistribution* analysis groups lines of source into different types of segments, quite similar to the *groupPerPhase* analysis, discussed in Section 6.2. A difference between these analyses is, that *segmentDistribution* returns the grouped source segments explicitly in the analysis result. To present this analysis result requires us to use the more specific presentation functions *renderStackBarChart* and *showAsTable2D*, which are fit for this result type. In some cases this approach can be favorable, because it provides more freedom in handling the analysis results. However, to calculate the relative values we now have to use an extra analysis, *relativeValueCount*, before the results can be presented using the previously mentioned presentation functions. In the previous analysis the *renderRelativeStackedBarChart* would have handled this. This presentation function would also rendered a slightly better presentation, for instance by setting a maximum of 100% for the range of the y-axis.

$$
\begin{aligned}
&\mathit{relmedianSourceLineDistribution} :: \mathit{AnalysisFK}\ \mathit{KeyHistory}\ [\mathit{Logging}]\ [(\mathit{SegmentType}, \mathit{Double})]\\
&\mathit{relmedianSourceLineDistribution} =\\
&\qquad \mathit{relativeValueCount}\\
&\quad <\cdot> \mathit{medianSegmentDistribution}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{relativeValueCount} :: (\mathit{DescriptiveKey}\ \mathit{key}, \mathit{Real}\ b, \mathit{Real}\ c, \mathit{Fractional}\ c) \Rightarrow \mathit{AnalysisFK}\ \mathit{key}\ [(a, b)]\ [(a, c)]\\
&\mathit{relativeValueCount} = \mathit{basicAnalysis}\ \texttt{"percentages"}\ \mathit{relativeCount}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{relativeCount} :: (\mathit{Real}\ b, \mathit{Real}\ c, \mathit{Fractional}\ c) \Rightarrow [(a, b)] \rightarrow [(a, c)]\\
&\mathit{relativeCount} = \ldots
\end{aligned}
$$

Analysis results

To study our claim we present the median size and the relative fractions of program segments for all students, per week in stacked bar charts in Figure 6.5 and the actual values in Figure 6.6. These figures are based on data from the 2004/2005 functional programming course.

The presentation of the median size of source segments, shown in the left-hand side of Figure 6.5, clearly shows an increase in the total (median) program segmentation for the weeks in which an assignment was to be handed in, week 9, 10 and 14. This clearly indicates increased activity, similarly as shown in the module length analysis in Section 6.1.

Overall we see that comments, or assumably documentation, is present in the logged programs in each week. In the first week, when programs are rather small, comments take more then 50% of the total source size. For the rest of the week the level of comments ranges between 15% and 25%. In the first few weeks these values decrease until the weeks in which assignments are to be handed in, week 9 and 10. (There were two first assignments, one for students of cognitive sciences and one for students of computer science). In the later weeks, week 11, 12, 13 and 14, the level of comments increases to about 25% with only an exceptional low value in week 12. It seems that for the second assignment student write comments throughout the development process, indicating a positive learning effect. The low value of comments in week 12 might be explained by the fact that there were no lectures and supervised labsessions that week due to an examination week, and only a small number of thirteen students worked in the computer rooms.

	CodeLn	CommentLn	EmptyLn
2005 6	7.0 (29.2%)	13.0 (54.2%)	4.0 (16.7%)
2005 7	17.0 (47.2%)	9.0 (25.0%)	10.0 (27.8%)
2005 8	40.0 (54.8%)	17.0 (23.3%)	16.0 (21.9%)
2005 9	66.0 (62.3%)	18.0 (17.0%)	22.0 (20.8%)
2005 10	68.0 (63.6%)	15.0 (14.0%)	24.0 (22.4%)
2005 11	36.0 (55.4%)	14.0 (21.5%)	15.0 (23.1%)
2005 12	42.0 (58.3%)	12.0 (16.7%)	18.0 (25.0%)
2005 13	61.0 (55.0%)	28.0 (25.2%)	22.0 (19.8%)
2005 14	62.0 (53.9%)	28.0 (24.3%)	25.0 (21.7%)

Figure 6.6: Median size and relative fractions of source program segments for all students, given per week from 2004/2005.

Conclusion and further analysis

The presented data in the previous section provides us a relatively clear view on the level of comments, and presumably the level of documentation. Comments are present throughout the period of the course and, when the first week is ignored, ranges between the 15% and 25%. Especially after the deadlines of the first assignments we see an increasing level of comments in the logged programs.

The studied period of 2004/2005 had different assignments (having different deadlines), based on the background of a student. It is possible that the analysis results from both groups are interfering. A future study could separate these groups to see whether they exhibit a different behavior.

Further research can analyze the difference in level of comments or documentation per student. Additional studies and less coarse preprocessing techniques are necessary to analyze whether comments contain documentation or just code, put in comments. Also the type signatures of functions can be counted as documentation. It is also interesting to analyze whether documenting early in the development process has a positive effect on the resulting program. This could be evaluated by inspecting the complexity of the resulting program or using the grade of the assignments, as an approximation.

6.4 Compilation interval analysis

Each student has his own approach to developing a program. A typical approach for developing functional programs is to write small extensions and compile the program to see whether the program works as expected. Typically, students programming Helium, use `hint` or `texthint`, a programming environment which provides an interpreter interface to the Helium compiler. These programming environments are very suitable for this way of working. More experienced programmers, interested in solving the problem faster, write more code before compiling again. From this we formulate the following claim:

Claim: Novice functional programmers compile at regular intervals. These average intervals between compiles increase over time.

What we consider to be a regular interval needs to be investigated, for this we use this analysis. By observing students working with Helium we expect this to be not more than 5 minutes.

Analysis design

The design of this analysis involves the notion of coherence as discussed in Section 4.2. Our aim is to find statistics that express the difference in compilation lengths. Typically we need to calculate the time between two consecutive loggings. However, we do need to set a maximum upper bound for the time allowed between two consecutive loggings. We do this because this prevents our presentations from becoming less expressive by extremes in the analysis outcomes. This could very well happen when a student works on his assignment in the morning, follows lectures, and continues to work on the assignment in the afternoon, or the next day. It is not likely that the student works on his program between the two occasions, so we need to separate these two programming sessions. For this we construct a time coherence analysis:

```
calcTimeCoherentSessions :: Int → AnalysisFK KeyHistory [Logging] [[Logging]]
calcTimeCoherentSessions min =
  basicAnalysis   ("time coherence of " ++ show min ++ " min")
                  (groupByCoherence (λl1 l2 → logTimeDiff l1 l2 < minutesOfTD min))
logTimeDiff    :: Logging → Logging → TimeDiff
logTimeDiff    = ...
groupByCoherence :: (Logging → Logging → Bool) → [Logging] → [[Logging]]
groupByCoherence = ...
minutesOfTD :: Int → TimeDiff
minutesOfTD = ...
```

The analysis *calcTimeCoherentSessions* groups the loggings using *basicAnalysis* in combination with the function *groupByCoherence*. The latter function groups loggings according the principle of coherence, discussed in Section 4.2, This function uses the first parameter to check whether two subsequent logging are related.

The analysis *calcTimeCoherentSessions* uses the primitive analysis *basicAnalysis* and not *groupAnalysis*, because this provides the freedom to handle the complete set of time coherent logging sequences in a later stage of the analysis. Furthermore, the upper bound for the time coherence is a parameter for this analysis.

With *calcTimeCoherentSessions* we construct *calculateCompilationTimeDiffs*, an analysis to calculate the difference in time between each pair of consecutive loggings from a complete sequence of loggings:

```
calculateCompilationTimeDiffs :: Int → AnalysisFK KeyHistory [Logging] [Float]
calculateCompilationTimeDiffs min =
      mapAnalysis (basicAnalysis "relative time difference" (timeDiffToMin . uncurry logTimeDiff))
  <·> concatMapAnalysis (basicAnalysis "tupling per two loggings" tuplePerPair)
  <·> calcTimeCoherentSessions min

tuplePerPair :: [a] → [(a, a)]
tuplePerPair input = zip (init input) (tail input)
```

The analysis *calculateCompilationTimeDiffs* first groups the loggings in time coherent sessions, using the upper bound provided as a parameter. Then the time coherent loggings are tupled two by two and for each tuple the difference in time is calculated.

The analysis *calculateCompilationTimeDiffs* only works for the loggings of a particular student. To assure this, we construct the analysis *compilationTimeDiff* which reuses *calculateCompilationTimeDiffs* but applies this analysis on the loggings from each student separately.

```
compilationTimeDiff :: Int → Int → AnalysisFK KeyHistory [Float] (OptData b)
                                 → AnalysisFK KeyHistory [Logging] [OptData b]
compilationTimeDiff minsize min aggregateTDAnaPerStudent =
      aggregateAnalysis "" id
  <·> aggregateTDAnaPerStudent
  <·> calculateCompilationTimeDiffs min
  <·> selectGrpAnalysis'_ ((>minsize) . length)
  <·> groupPerStudent
```

The analysis *compilationTimeDiff* first groups the loggings per student, then the analysis selects the loggings of students which have at least *minsize* loggings. (*minsize* is provided as parameter.) We perform this selection so we can avoid students who are hardly active and just use Helium to try some small examples (e.g. just to check a function type), without actually programming with the purpose to solve a particular exercise. Next the *calculateCompilationTimeDiffs* analysis is applied. The result of this analysis is then aggregated, per student, with an aggregation analysis, again provided a parameter. As a final analysis step the outcome is aggregated into a single result set, using the *aggregateAnalysis*. Concrete analyses which calculate the average and median compilation interval per student is defined by *avgCompilationTimeDiff* and *medianCompilationTimeDiff*. These analyses are used to generate box plots, discussed in the next section, using the *renderBoxPlot* presentation function.

```
avgCompilationTimeDiff    = compilationTimeDiff 10 60 averageAnalysis
medianCompilationTimeDiff = compilationTimeDiff 10 60 medianAnalysis
```

A downside to the presented implementation is that generating analyses for a grouping, say per week, now requires special care. This is because the analysis groups and aggregates the analysis result per student. To be able to render a presentation per week we now need to use the <:> analysis combinator. This combinator applies the analysis on each group separately.

$$
\begin{aligned}
&avgCompilationTimeDiffPerWeek = \\
&\quad avgCompilationTimeDiff \\
&<:> groupPerWeek
\end{aligned}
$$

Analysis results

Using the analysis implementation from the previous section we calculate the typical, meaning the average and median, compilation interval length per student, per week from 2003/2004. The data is presented using box plots, shown in Figure 6.7.

For the analysis we choose an upper bound of 60 minutes for the time coherence analysis and a minimum of ten loggings per week per student. Furthermore, we aggregate the compilation interval lengths per student, by taking the average and the median function, resulting in the two figures. The motivation for using two different aggregation functions is that the median, unlike the average, is not easily effected by extremes in the analysis results. Therefore the median is often considered as a better value to express central tendency. Nonetheless both figures tell us something about the compilation intervals of the students.

Figure 6.7 shows, as explained, higher average compilation interval lengths than median compilation interval lengths. The number under the axis show how many students were actually involved in the analysis calculation of that week. From the box plots we deduce that in most weeks 75% of the students have an average compilation interval length of at most four minutes, only week 10, 11 and 13 show slightly higher values. We can see this, because the top of a box represents the third quartile of the data set. 95% (the upper connected extreme of a boxplot) of the students have an average under the seven minutes. The median values are lower. 75% of the students have a compilation interval length lower then 1.5 minutes, and 95% lower then 2.5 minutes, except for week eleven and twelve. Week eleven and twelve show a strong increase in the median of compilation interval lengths for certain students.

Surprising in both presentations are the low average and low median values. Overall, one could say that students compile a subsequent program in average within 2.5 minutes and with a median of one minute! This certainly classifies as compiling at regular intervals, as described by our claim. However, from this data we cannot conclude whether these values are steadily increasing. Only the average compilation interval lengths show a moderate increase in the course of time.

Conclusion and further analysis

This analysis studied compilation interval lengths per student. We clearly see that students compile frequently, most students in average within 2.5 minutes. Unlike expected, this value, averaged over all students, does not significantly decrease. This could however still hold for certain students, but this needs more research.

A slight improvement to this analysis is to add the notion of content-coherence, which would require the subsequent loggings to be related. In our current implementation, module imports can have an undesirable effect on the (average) compilation interval length. Module imports, which are edited or for which no compiled version is available, are automatically compiled shortly in time before the parent module. By requiring that subsequent programs have the same filename, we prevent the program loggings, representing the compilation of module imports, to be included in the analysis. However, we expect not much change in our analysis results, since module imports do not occur often in our studied data set.

Further research could also include the effect of the compiler responses on the compilation interval length. Compilation intervals for more complex phases like the type inferencing phase might take the programmer more time to come up with a solution, resulting in a longer compilation interval.

Other directions of research involve combining this study with an analysis which analyzes the change in the source of consecutive logged programs. By studying the difference between consecutive programs, one can analyze the number of added, changed, or removed program elements. This change can be expressed in added, changed or deleted code lines, words or nodes from an abstract syntax tree. These analyses, in combination

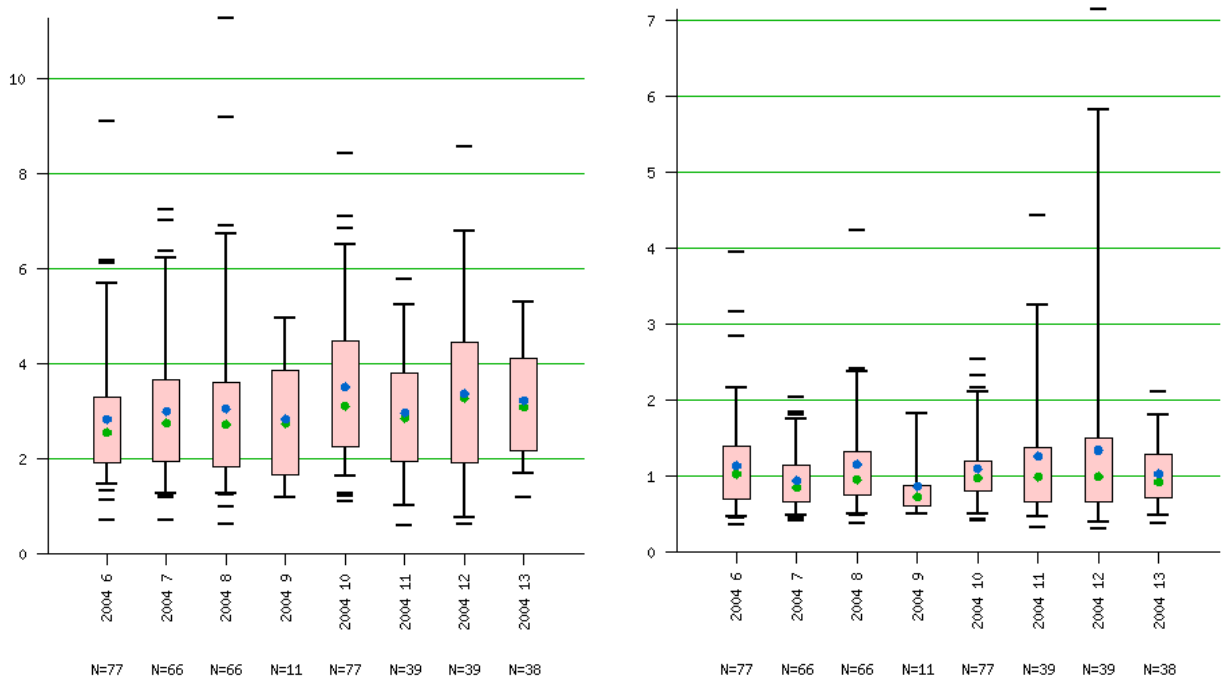


Figure 6.7: Average (left) and median (right) compilation intervals (in minutes) for all students, given per week with a time coherence of 60 minutes from 2003/2004.

with the compilation interval analysis can tell us something about the style of programming of a novice programmer.

6.5 Type error repair analysis

Type errors are in average the most occurring errors for students programming Helium as shown by the phase analysis in Section 6.2. An interesting analysis is to analyze how long and how much effort, expressed in number of compiles, it takes for a student to construct a type correct program from a parseable, but type incorrect program. Through experience students are expected to learn to use a programming language. An aim in learning a functional programming language, is learning to handle the sophisticated type system. In time, one would expect a student to be able to resolve type errors faster than in the beginning of a programming course. On the other hand, we also expect that programs become more complex, resulting in harder to resolve type errors. With this case study, we analyze the following claim:

Claim: The time and effort (expressed in required number of compiles) needed to solve a type error, decreases over time.

Analysis design

To be able to analyze the time and effort needed to repair a type incorrect program, we must first specify how we can detect the process of correcting a type incorrect program. We do this by locating a first type incorrect logging in a sequence of loggings, from a particular student. Then, we determine the related and subsequent loggings generated by the student until we arrive at a correct compile. In the rest of this document we refer to these sequences of loggings as *type correcting sequences*. By studying the average number of loggings and the time between the first and the last logging of such sequences of loggings, we are able to study whether these values change over time. Both these analyses are very similar, but are slightly different in the way they use the notion of coherence.

To calculate the number of compiles, we start by computing, for a given student, the sequence of loggings that deal with the same program file. We do this because we do not want that a correct compile of another

module is viewed as a repair. This is in fact an implementation of the notion of content-coherence as discussed in Section 4.2. This analysis is implemented by the *calcFilenameCoherentSequences* analysis.

```
calcFilenameCoherentSequences :: AnalysisFK KeyHistory [Logging] [[Logging]]
calcFilenameCoherentSequences =
  basicAnalysis "coherence grouping on: filename"
                (groupByCoherence sameSourceFile)
groupByCoherence :: (Logging → Logging → Bool) → [Logging] → [[Logging]]
groupByCoherence = ...
sameSourceFile :: Logging → Logging → Bool
sameSourceFile = ...
```

A next step in the analysis is to calculate the loggings which represent the process of repairing a type incorrect program, the so-called type correcting sequences. We do this by breaking a sequence of loggings into smaller sequences that start with a type error and end in a correct compile, as described in the beginning of this section. For example, given the following sequence of phases of loggings (denoted by their phase) $[C\ P\ T\ T\ T\ P\ T\ C\ L\ L\ P\ P\ T\ C\ P]$, we obtain two sequences $[T\ T\ T\ P\ T\ C]$ and $[T\ C]$. This analysis is defined by *calculateTCSequencesAnalysis*.

```
calculateTCSequencesAnalysis :: AnalysisFK KeyHistory [Logging] [[Logging]]
calculateTCSequencesAnalysis =
  basicAnalysis "Group of loggings from TtoC" calculateTCSequences
calculateTCSequences :: [Logging] → [[Logging]]
calculateTCSequences = ...
```

The analysis *countNumberOfLoggings* can be used to compute the number of compiles, as discussed in Section 6.2. To compute these values per student from a complete logging data set using these analyses can be done similar to *compilationTimeDiff*, an analysis discussed with the compilation interval analysis, in Section 6.4. In the end of this section we present a method to analyze this aspects per student.

To compute the time needed to solve a type error we reuse the *calculateTCSequencesAnalysis* analysis, and extend the coherence analysis. We now intend to measure the time needed to solve, so we also use the notion of time-coherence. The analysis *calcTimeAndFilenameCoherence* is constructed for this purpose.

```
calcTimeAndFilenameCoherentSequences :: Int → AnalysisFK KeyHistory [Logging] [[Logging]]
calcTimeAndFilenameCoherentSequences min =
  basicAnalysis ("coherence grouping on: filename and time range (" ++ show min ++ ")")
                (groupByCoherence
                  (λl1 l2 → sameSourceFile l1 l2
                    ∧
                      logTimeDiff l1 l2 < minutesOfTD min
                  ))
```

Note that we split the sequences of loggings into time coherent programming sessions to avoid the situation that a student fixes a problem two days later. This would certainly muddy the waters, because he is probably not working on solving the problem during this entire period. A consequence of this is that the resulted sequences of loggings are slightly different compared to the sequences computed for the number of compiles analysis (based on only content-coherence). The consequences of this will be discussed in the conclusion section.

To calculate the time needed to correct a type incorrect program, we construct the *timeTCSequences* analysis. This calculates the difference in time between the first logging and the last logging of a sequence of loggings, representing the process of repairing a type incorrect program.

```
timeTCSequences :: AnalysisFK KeyHistory [Logging] TimeDiff
timeTCSequences = basicAnalysis "" (uncurry logTimeDiff . headlast)
logTimeDiff ::    Logging → Logging → TimeDiff
logTimeDiff =     ...
headlast :: [a] → (a, a)
headlast list = (head list, last list)
```

The analyses *timeDiffToSecAnalysis*, *timeDiffToMinAnalysis*, or similar analyses can be used to compute the difference in time for a certain time unit.

```
timeDiffToSecAnalysis :: AnalysisFK KeyHistory TimeDiff Integer
timeDiffToMinAnalysis :: AnalysisFK KeyHistory TimeDiff Float
```

To calculate the time needed to repair a type incorrect program per student from a complete logging data set, we can combine the presented analysis, in a similar fashion as for *compilationTimeDiff*, an analysis discussed with the compilation interval analysis, in Section 6.4. The result of these analyses can be shown using a box plot presentation. When using an aggregation function, like average of median, the results can be presented in bar charts, and tables.

To run an analyses per student, one can use the *filterAnalysis* or *splitAnalysis* combinator. The *filterAnalysis* allows to select only the loggings from one particular user (after applying a grouping analysis, for instance per student). The *splitAnalysis* handles loggings of each student separately. For instance, to calculate the number of compiles per student, one can define:

```
analysisPerStudent :: AnalysisFK KeyHistory [Logging] a → [(KeyHistory, [Logging])] → [[(KeyHistory, a)]]
analysisPerStudent ana = map ana . splitAnalysis . groupPerStudent
numOfCompilesPerStudent :: Int → [(KeyHistory, [Logging])] → [[(KeyHistory, [Int])]]
numOfCompilesPerStudent minsize = analysisPerStudent
    (   mapAnalysis countNumberOfLoggings
  <·> concatMapAnalysis calculateTCSequencesAnalysis
  <·> calcFilenameCoherentSequences
  <·> selectGrpAnalysis'_ ((>minsize) . length)
  <·> groupPerWeek)
```

Due to the type of *splitAnalysis*, the analysis *numOfCompilesPerStudent* and the result of *analysisPerStudent* are not of the common *AnalysisFK key a b* type. To present such an analysis, one now has to map a presentation function over the set of analysis results.

Analysis results

For this study we generate the average number of compiles and the average time needed for each student, and display this information per week for all students from the 2003/2004 logging data set in box plot diagrams, shown in Figure 6.8. We leave the variant of this analysis, which uses the median function to aggregate per student, for future analysis. For all presentations displayed in this section, we used a minimal size of at least ten compiles per week, to assure that presented data is based on a reasonable amount of compiles. Furthermore, we ignore week 9 in our discussion, because it is a holiday week and is not representative enough.

The left-hand side of Figure 6.8 shows the average number of compiles needed, per student, to repair a type incorrect program. The minimum number of compiles required to repair a type incorrect program is two, which services as baseline of this metric on the y-axis.

In the first few weeks, week 6 to 10, no clear change in the distribution of the average values is visible. In these weeks, 50% of the students repair a type incorrect program in average in 3 to 5 compiles. In these weeks the students work mostly on simple exercises and the first graded assignment. In the weeks 11 to 13 we see a different development. In these weeks students work on more complex exercises and the second assignment. In this period, the overall average number of compiles decreases, with 50% of the students needing in average 3 to 4 compiles, in week 13. Also, high average values occur less often. A remark to these observations is that the number of students involved in the period from week 11 to 13, shown by the N-number under the x-axis, is roughly the half of the number of students in the period of week 6 to 10.

The right-hand side of Figure 6.8 shows the time (in number of seconds) needed to repair a type incorrect program. This analysis also requires the compilation to be time-coherent. For this we choose the time parameter to be ten minutes, so consecutive loggings are not more than ten minutes apart. Although we did not (yet) thoroughly investigate the aspect of programming sessions, we know from the compilation interval analysis in Section 6.4, that most compilations are not more than ten minutes apart. As such we expect to cover most programming sessions with this value. However, as a consequence of this requirement, there are slightly less students for which we can compute the type correcting sequences, in week 8 and 10. Further analysis, showed that in total (for all students) the number of type correcting sequences, on which these box

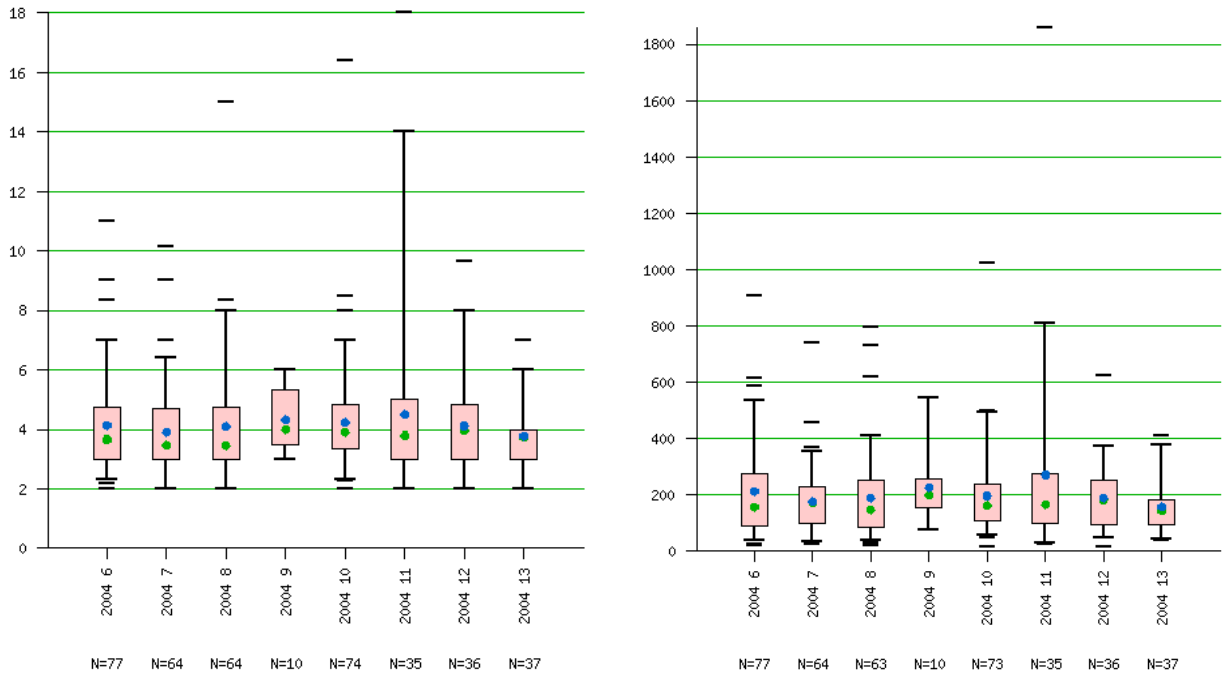


Figure 6.8: Average number of compiles (left) and average time (right, in seconds, 10 minutes time coherence) needed to repair a type incorrect program for all students, from 2003/2004.

plots are based, range from 175 to 780 per week, in average about 6.5 type correcting sequences per student. When the time-coherence is involved in the analysis, the total number of type correcting sequences drop in average with 14 cases per week, about 2%.

When analyzing the weekly average times needed to repair a type incorrect program we see a similar development as with the average number of compiles analysis. The first few weeks, week 6 to 10, do not show a particular positive of negative progression. In the period of week 11 to 13 we see a slight, but steady decrease in the number of required compiles. From this presentation we can conclude that 75% of the students are able to repair a type incorrect program in less then 300 seconds (5.0 minutes). In the last week this is even less than 200 seconds (3.3 minutes).

We are also able generate these presentations per student. Unfortunately, for most students are not enough loggings available per week to render a proper presentation. We limit our analysis to student 117 and student 120, as shown in Figure 6.9 and Figure 6.10. Both students are very active in working with Helium. In these presentations the numbers under the x-axis tell us the number of type correcting sequences, which have been computed.

For student 117, we only study week 9 to 12, because these weeks deliver a reasonable number of type correcting sequences. In these weeks the student uses in average 5 compiles and roughly about 200 seconds to repair a type incorrect program. With these values the student shows the average behavior, as seen in the previous overall presentations. However, we are not able to detect a decrease in the values for week 11 to 13, as shown in the previous presentations.

For student 120, we see a different behavior. The number of compiles, increases over time, with 2.5 compiles in week 7 and 8, and in average around 3.5 in the later weeks. However, the average time needed to repair a type incorrect program shows a decrease in week 10 to 12, something we also saw for the complete group of students.

Conclusion and further analysis

From the presented results of this analysis, we are not able to deliver strong evidence which would prove our claim, regarding an overall decrease of the required number of compiles (effort) and time needed to repair a type incorrect program. We do however, notice a decrease of these values in the last three weeks, but are not able to isolate this separately, for a significant number of students. Also the fact that only half the number

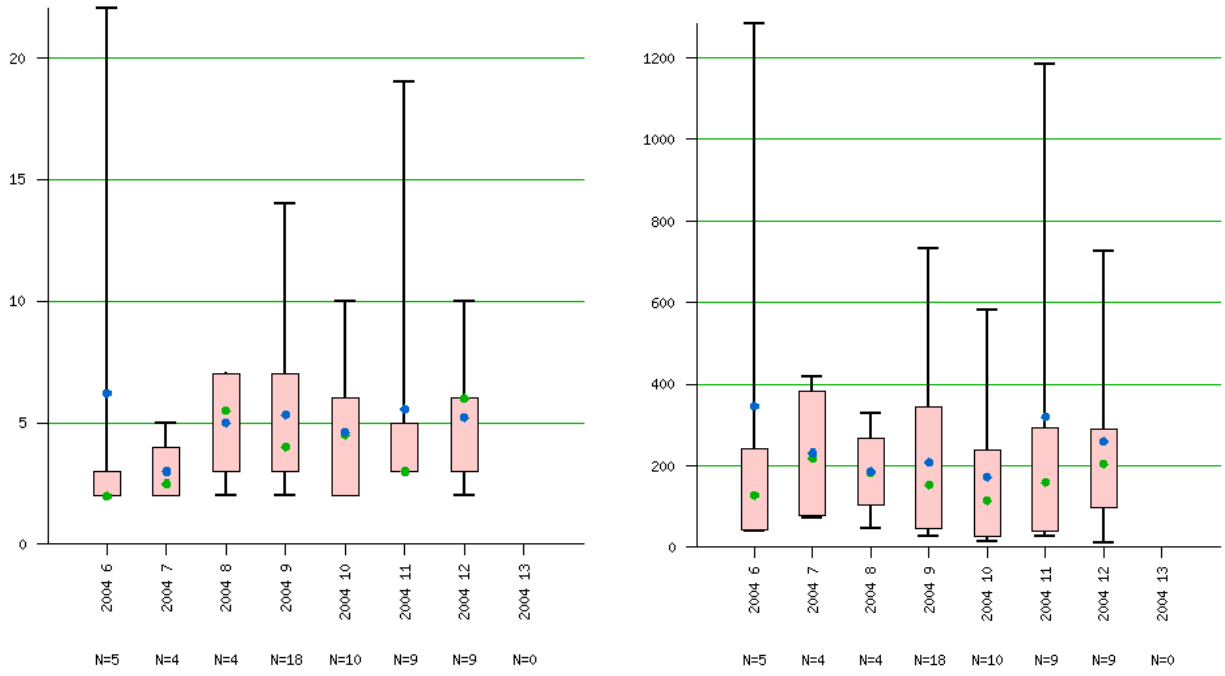


Figure 6.9: Number of compiles (left) and time (right, in seconds, 10 minutes time coherence) needed to repair a type incorrect program for student 117, from 2003/2004.

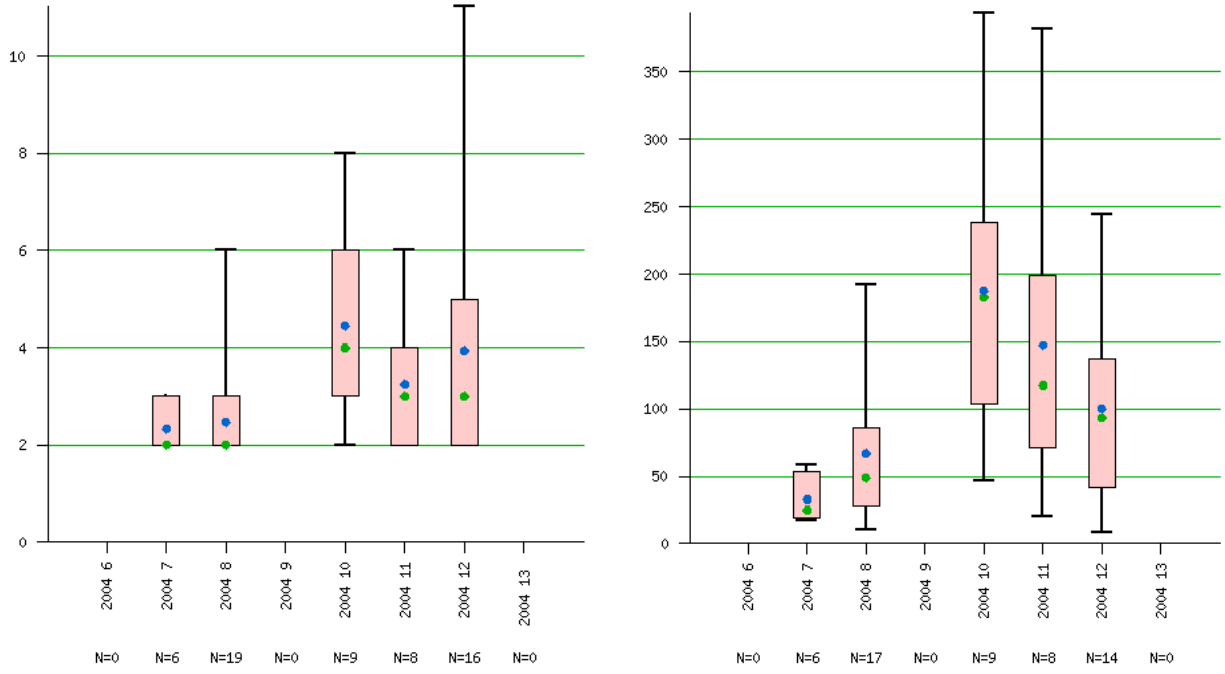


Figure 6.10: Number of compiles (left) and time (right, in seconds, 10 minutes time coherence) needed to repair a type incorrect program for student 120, from 2003/2004.

of students are found in the data set for this period, compared to the first weeks, could strongly influence our results. It could be the case that a group of good programmers continues the course, where slow programming students quit the course. A future research, could focus on the continuing group, to see whether this group shows a decrease for the time and effort needed to repair a type incorrect program.

Further research could study the aspects which influence the number of compiles and amount of time. For instance, the kind of type error or the complexity of the type error could be evaluated. Also the effect of informative hints could be studied. The latter will be done in Section 6.7.

Other (similar) directions of research could study the effort and time needed to repair a lexical incorrect program, an unparsable program, or program which fails in the static analysis phase. Also the complete process of developing a runable program, so computing the number of compiles and needed time to bring a program through the stages of lexical analysis, parsing, static analysis and type inferencing all together, could be subject of study.

6.6 Type hints analysis

The Helium compiler is capable of generating high quality error messages, in comparison to other Haskell compilers, like GHC. In addition to an error message, hints and warnings are generated which help the student in solving errors. Each phase provides hints, which are meant to guide the programmer in the development process. The typing phase is often regarded as the hardest and most complex phase. Assisting students to pass this phase, clearly helps to effectively construct programs. Much effort is therefore spent on the type inferencer of Helium and additional techniques to generate helpful hints for a programmer. For instance, a programmer is told to use the sibling function (:) instead of (++), if this resolves the typing error.

The question that one might ask, is how many type error messages are actually accompanied with a hint and how this ratio of hints, expressed as percentage, changes over time. We expect this value to decrease, because students are supposed to learn how they can deal with type errors, especially when a hint is provided with the error. From this we formulate the following claim, which we will study in this analysis.

Claim: A significant fraction of type error messages are accompanied with a hint and this value decreases over time

Analysis design

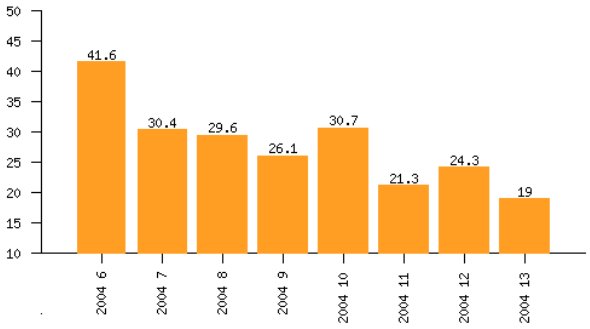
To implement an analysis calculating the number of loggings with or without a type hint is relatively easy. The general approach is to select the loggings for which the compilation process ended in the typing phase. For this we compute the total number of loggings and the number of loggings with a hint. The detection of a hint is easily achieved by applying a regular expression matching on the text which is prefixed with the actual hint text. In Figure 2.1 we see that this text is "`probable fix  :`". With this knowledge, we construct *typeHintsAnalysis*, an analysis performing a textual analysis, returning the type hints, each presented with one of the type errors.

```
typeHintsAnalysis :: AnalysisFK KeyHistory Logging [String]
typeHintsAnalysis =
  basicAnalysis "type hint(s)" (concat . multipleMatchRegex hint_regex . compileroutput)
  where
    hint_regex = mkRegexWithOpts " *(probable fix.*)" True True
multipleMatchRegex :: Regex → String → [[String]]
multipleMatchRegex = ...
```

The function *compileroutput* is a helper function for handling loggings and is discussed in Appendix E.2. The function *multipleMatchRegex* applies the regular expression repeatedly to a string, returning all matching subexpressions.

We define *countNumberOfLoggingsWithHint* to count the number of hints in a sequence of loggings.

```
countNumberOfLoggingsWithHint :: AnalysisFK KeyHistory [Logging] Int
countNumberOfLoggingsWithHint =
```



	Hint ratio
2004 6	41.6% (446 / 1073)
2004 7	30.4% (363 / 1195)
2004 8	29.6% (318 / 1074)
2004 9	26.1% (85 / 326)
2004 10	30.7% (580 / 1890)
2004 11	21.3% (101 / 474)
2004 12	24.3% (149 / 612)
2004 13	19.0% (122 / 641)

Figure 6.11: Hint ratio for type incorrect compiles, from 2003/2004, given per week.

```
    basicAnalysis "number of compiles with hint" (length . filter (not . null))
<·> mapAnalysis typeHintsAnalysis
```

This analysis counts the compilations with at least one type hint. It may happen that there are several type errors in the compiler feedback, not all necessarily with a hint. We assume that students work on one error at a time and that the student is therefore helped if at least one hint is provided. As such, the compiler feedback guides the student in solving the programming exercises.

To calculate the number of compiles with and without a type hint we construct *numberOfTypeHintsAnalysis*. This analysis first selects the type incorrect compiles, and then applies *countNumberOfLoggings*, presented in Section 6.2, and *countNumberOfLoggingsWithHint*, in parallel to calculate the total number of compiles and the number of compiles with at least one hint. The analysis *percentageOfTypeHintsAnalysis* extends this analysis, by calculating the ratio (percentage) from these values, and encapsulating the percentage and the values from which the percentage is calculated, in a custom data type *Percentage*. This custom data type allows a more flexible presentation.

```
numberOfTypeHintsAnalysis :: AnalysisFK KeyHistory [Logging] (Int, Int)
numberOfTypeHintsAnalysis =
        (countNumberOfLoggings <&> countNumberOfLoggingsWithHint)
    <·> basicAnalysis "(from only type incorrect programs)" (filter ((≡ Typing) . phase))
percentageOfTypeHintsAnalysis :: AnalysisFK KeyHistory [Logging] Percentage
percentageOfTypeHintsAnalysis =
        basicAnalysis "percentage" (λ(total, x) → Percentage (percentage x total) x total)
    <·> numberOfTypeHintsAnalysis
data Percentage = Percentage Double Int Int
percentage :: (Real a, Real b, Fractional b) ⇒ a → a → b
percentage = ...
```

Analysis result

Using the presented analysis, we are able to compute the percentage of type hints for a complete data set, as well as per week. For the 2003/2004 data set, we compute that 29.7% of the type incorrect compiles, is presented with at least one hint. This value is based on 2,164 type incorrect compiles with at least one hint, in relation to a total of 7,285 type incorrect compiles, for the complete data set. This is quite a high number, so almost a third of all type error compiler responses, is presented with at least one hint.

The percentages per week are shown in Figure 6.11. Again we ignore week 9 in our discussion, because this is a holiday week. From the bar chart on the left-hand side of this figure, we see a clear drop of more then 10% after the first week. For the weeks 7, 8, and 10 this value remains around 30%. In week 11, the week after the deadline of the first assignment, we see another decrease of 10%. Week 12 shows a slight increase of 3%. Week 13 this value is about 20%. Overall we clearly see a decreasing ratio of type hints, indicating that students make fewer type errors, for which Helium can generate an informative hint.

Conclusion and further analysis

This analysis provides a clear view on the ratio of type hints and the change of this value over time. For the complete data set of 2003/2004 we see a hint ratio of about 30%, a value we consider to be significantly high. The hint ratio decreases over time, from about 40% in the first week, to about 20% in the last week. This indicates that students have learned to avoid these mistakes. Further research is necessary to study whether this value stabilizes over time (optionally also for students separately), which kind of type hints are most common, and which type of hints especially are presented less or more over time.

6.7 Effectiveness of hints analysis

Errors are often accompanied with a probable solution, a hint. These hints are meant as guidance in the process of programming. With time and effort spent on advanced techniques to produce such hints, one wonders what the effect is on a programmer. For instance, what is the effect on the time needed to solve a type error. For this analysis we formulate the following claim:

Claim: The development process of repairing a type incorrect program, benefits significantly from the ratio of type hints presented during this process.

This analysis, focuses specifically on solving type errors, because for this situation we can conveniently reuse the analyses presented in Section 6.5 and Section 6.6. We primarily focus on the time needed to repair a type incorrect program.

Analysis design

This analysis evaluates the effect of hints on the time needed to solve a type error. We can visualize this effect by plotting the ratio of type hints (the factor of interest), discussed in Section 6.6, on the x-axis, and the time needed to repair a type incorrect program (the response variable), discussed in Section 6.5, on the y-axis. As such, we hope to see that when the ratio of type hints, increases, the time needed to repair a type incorrect program decreases.

The ratio of type hints is calculated by counting how many type errors during the process of repairing a type incorrect program is presented to the programmer with a type hint, and how many type errors are presented in total. We do this because we assume that a student is working on one error at a time, and he or she is helped with at least one hint for one of the type errors. As such, we assume the student is mostly likely to solve the type error for which a type hint is available.

In addition to this analysis, we also compute the number of compiles, to provide more insight into the number of compiles, related to the calculated repair times.

For this analysis we use complete logging data sets, because there are hardly any students with enough logged compiles from which we can compute a reasonable number of type correcting sequences per week.

We start by constructing *hintEffectivenessTimeAnalysis*, an analysis which calculates the ratio of compiler responses with at least one type hint and the time needed for a programmer to repair a type incorrect program.

```
hintEffectivenessTimeAnalysis :: Int → AnalysisFK KeyHistory [Logging] [(Double, Integer)]
hintEffectivenessTimeAnalysis min =
      mapAnalysis (percentageOfTypeHintsAnalysis' <&> (timeDiffToSecAnalysis <·> timeTCSequences))
  <·> concatMapAnalysis calculateTCSequencesAnalysis
  <·> calcTimeAndFilenameCoherentSequences min

percentageOfTypeHintsAnalysis' :: AnalysisFK KeyHistory [Logging] Double
percentageOfTypeHintsAnalysis' = ...
```

The analysis *hintEffectivenessTimeAnalysis* consists of several smaller analyses, introduced in previous sections. First the analysis *calcTimeAndFilenameCoherentSequences* and *calculateTCSequencesAnalysis*, introduced in Section 6.5, are used to break the sequence of loggings into sessions and break these sessions into type correcting sequences.

Next, the analysis *percentageOfTypeHintsAnalysis′* is used to calculate the percentage (ratio) of compiler responses with at least one type hint presented with one of the type errors. This analysis is a variant of the *percentageOfTypeHintsAnalysis* analysis from Section 6.6, returning the percentage in a *Double* instead of the custom *Percentage* data type. The analyses *timeDiffToSecAnalysis* and *timeTCSequences*, discussed in Section 6.4, are used to calculate the time between the first and the last logging in the type correcting sequences. The analyses are applied tuplewise so the resulting set of data points can be presented using a scatterplot.

Similarly, we also construct *hintEffectivenessCompilesAnalysis*, an analysis which calculates the number of compiles, required to repair a type incorrect program. This analysis uses *countNumberOfLoggings*, discussed in Section 6.6 to count the number of compiles in type correcting sequences. To compute the programming sessions, we use *calcFilenameCoherentSequences*, as presented in Section 6.5. The result of this analysis can also presented as a scatterplot.

```
hintEffectivenessCompilesAnalysis :: AnalysisFK KeyHistory [Logging] [(Double, Int)]
hintEffectivenessCompilesAnalysis =
        mapAnalysis (percentageOfTypeHintsAnalysis′ <&> countNumberOfLoggings)
    <·> concatMapAnalysis calculateTCSequencesAnalysis
    <·> calcFilenameCoherentSequences
```

Currently, a scatterplot presentation is not available in our descriptive statistical library, so to present the calculated times to repair a type incorrect program, we resort to another presentation, the box plot diagram. By dividing the complete range of type hint ratios (from 0% to 100%) in intervals of each ten percent, we create a categorical system in which each data point maps to one of the intervals. So a hint ratio of 0% maps to the first interval of 0% to 9%. A hint ratio of 15% maps to the second interval of 10% to 19%, and so on. So in our presentation we get ten box plots, placed in the center of each interval. Each such box plot represents a set of calculated times needed to repair type incorrect programs, for which a certain interval of hint ratios was available.

To present the data using a box plot presentation, we use *calculateHintEffectiveness*. This analysis first groups the loggings per student and per week, so we are able to set minimum amount of compiles per student, per week. After applying *hintEffectivenessTimeAnalysis* we combine the result in one result set, by using the *aggregateAnalysis* primitive. Next, we group the result in intervals of ten percent each, as described in the previous paragraph, using *categorizeValuesAnalysis*. As a final analysis step we flatten the analysis result to be able to render the box plot presentation.

```
calculateHintEffectiveness :: Int → Int → AnalysisFK KeyHistory [Logging] [Integer]
calculateHintEffectiveness min minsize =
        flattenAnalysis′
    <·> categorizeValuesAnalysis
    <·> aggregateAnalysis "hint effectiveness in time" concat
    <·> hintEffectivenessTimeAnalysis min
    <·> selectGrpAnalysis′_ ((>minsize) . length)
    <·> groupPerStudent
    <·> groupPerWeek
categorizeValuesAnalysis :: AnalysisFK KeyHistory [(Double, b)] [(Int, [b])]
categorizeValuesAnalysis = basicAnalysis "" categorizeValues
categorizeValues :: [(Double, b)] → [(Int, [b])]
categorizeValues = ...
```

A similar analysis is also constructed to calculate the correlation between the hint ratio and the number of compiles, based on *hintEffectivenessCompilesAnalysis*.

Analysis result

Running the analyses on the logging data set of 2003/2004 results in the plots shown in Figure 6.12. For these diagrams we use a minimum of ten loggings per student per week, and a time-coherence of ten minutes for the diagram on the left-hand side.

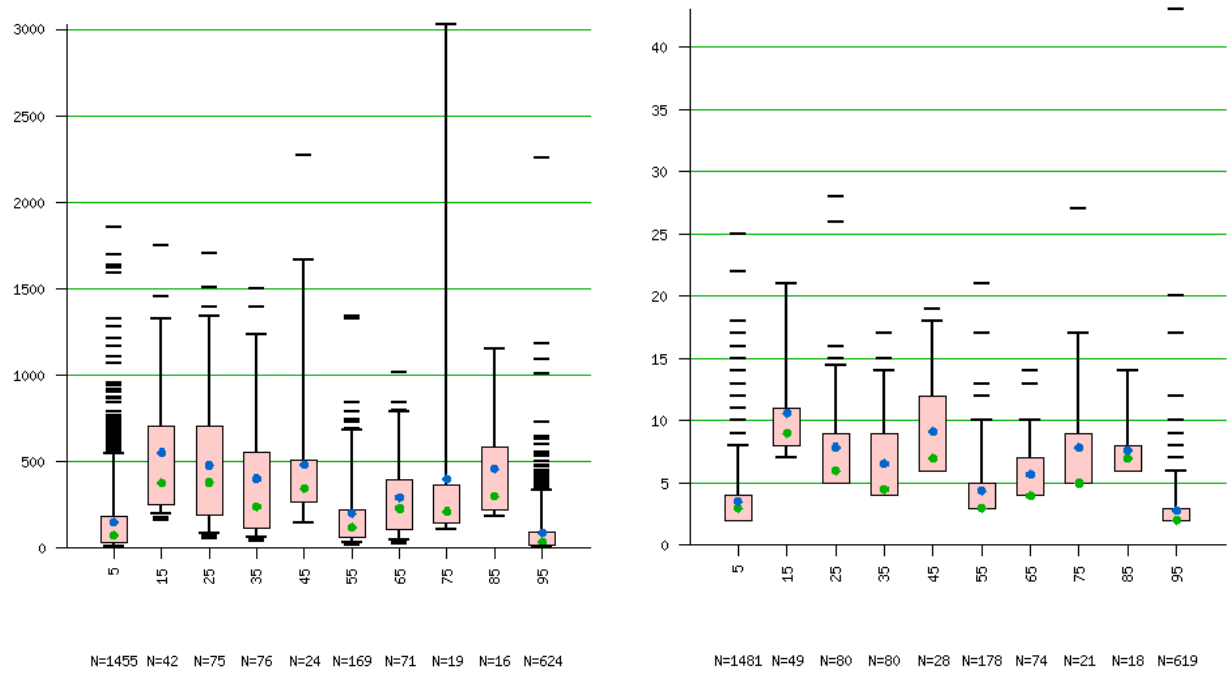


Figure 6.12: Correlation between ratio of type hints with time (left) and number of compiles (right) needed to repair a type incorrect program, from 2003/2004.

Both diagrams show, based on the high number for N (shown under the x-axis), that most type correcting sequences have a hint ratio in the range of either 0-9% or 90-100% percent. From the right-hand side of Figure 6.12 we deduce that this is mostly the case for very simple type correcting sequences, which just take about 2 or 3 compiles to solve the type error. These two box plots show, for these two particular situations, that when hints are provided with the type error(s), the time needed to repair a type incorrect program is roughly half of the time, then when no type hint is provided at all.

The left-hand side of Figure 6.12 does not show a clear and steady decrease. For instance, the time needed to repair a type incorrect program for which a hint ratio of 80-89% in computed, takes more time than the cases for which a hint ratio of 70-79% is computed. A possible explanation for this might be the fact that the ratios which fall in the 80-89% range, most likely 80% and 85%, also require more compiles (minimal 5 or 10 compiles) thus more time to solve. On the other hand, a hint ratio of 75%, which falls in the range of 70-79%, has a minimum of 3 compiles. This example reveals an unpleasant relation between the hint ratio and the time needed to repair a type incorrect program.

Nonetheless, the presentation seems to provide a view in which the average time needed to repair a type incorrect program, takes less time when about more then 50% of the compiler responses presenting type error to the programmer are combined with one or more informative hints. This becomes even more clear when we calculate the weighted average for the median values per interval, leaving out the 0-9% and the 90-100% interval. The weighted median average for the 10-49% range is 327.9 seconds (5.5 minutes). For the 50-89% this is 167.7 seconds (2.8 minutes). So having a hint ratio of more then 50%, cuts the needed time to repair a type incorrect program, in average, in half.

We must be careful interpreting these results, because it is likely that the cases of type correcting sequences which have a hint ratio of 50%, biases our reading, as explained previously. (These cases are a lot more frequent, 169 sequences fall in the 50-59% interval). When we leave out this interval and calculate the weighted average median for the 60-89% range, this value is 238.9 seconds (4.0 minutes). This value is just 0.73 times less then the weighted average median for the 10-49% range, this improvement is about 90 seconds (1.5 minutes). This still holds as a significant improvement.

Conclusion and further analysis

The presented results from this analysis do not show the steady overall decrease of the time needed to repair a type incorrect program we hoped for. Nonetheless, the presentation of time needed to repair a type incorrect program does provide evidence that indicates that a programmer significantly benefits from type hints, especially when more than 50% of the compiler responses presenting type errors, is combined with one or more hints.

Further analysis can evaluate whether the benefit of type hints also holds for separate students and whether this also holds over course of time. From the analysis of type hint ratios over time in Section 6.6, we know that the ratio decreases over time. This could be explained by students learning not to make certain type errors anymore. This indicates that the helpful effect of type hints decreases over time, possibly resulting in type correcting sequences which do not benefit (as much) from type hints.

Also could a different analysis approach be taken, in analyzing which type error a student is trying to solve. One could for instance, count the number of type error found in the first type error of the type correcting sequences. Dividing the required time and the number of compiles with this number might yield different interesting analysis results, focusing more on the first detected type error, and less on the overall process of solving a type error. Also could the ratio of hints, provided with the first type incorrect program of a repair sequence, be studied for effectiveness.

Chapter 7

Conclusion and future work

This thesis discussed the process of analyzing Helium logging data sets. Our main contribution in this research is the development of a tool set, which supports the process of (incremental) empirical research on collected logging data sets. The major component in this tool set is a statistical combinator library, which provides supports for constructing analyses in a modular fashion. Furthermore, we introduced notions for partitioning sequences of loggings, based on difference in time or content between two or more subsequent loggings. With these abstract notions of coherence, we are able to analyze aspects which involve sequences of related loggings, rather then just single loggings. Coherence plays for instance an important role when analyzing the time needed to solve a type error. The notions of coherence and descriptive statistics are also applicable in a broader sense. Other compilers, and similar software tools, if equipped with a logging facility, can easily reuse these concepts.

A practical part of this thesis is the preprocessing of collected data sets obtained from three incarnations of teaching functional programming to novice programmers and the utilities that derive from this task. In addition, we also collected eleven (sub)versions of Helium, used throughout the courses. With this set of compilers, the original compiler output has been reproduced for further analysis.

To show the applicability of our research and our tool set, we presented a number of representative case studies in Chapter 6. These analyses provide a first view on valuable information hidden in the data sets. More research is necessary to establish a more in-depth view on the studied programming aspects.

As a conclusion to this thesis, we present in this chapter possible next steps in research. We discuss possible areas of research regarding Helium loggings in Section 7.1. In Section 7.2 and Section 7.3 we discuss future applications of our tool set specifically for programmers and lecturers.

7.1 Evaluating Helium

An obvious research direction is the evaluation of Helium as a functional programming language as well as a compiler. The language represents almost completely the `Haskell 98` standard. Observations in the use of Helium are probably also applicable to Haskell. Interesting research questions relate to source program characteristics and programming style. Examples of source program characteristics are module length, compilation phase, and use of language constructs. Programming style analyses involves also what happens between compilations, for example one can study compilation frequency (the time between loggings) and programming efforts between compiles. Some of these aspects have already been studied in Chapter 6. However, more in-depth and improved analysis will provide better understanding of these aspects.

Evaluating Helium as a compiler should also consider the analysis of errors, warnings, and hints presented to a programmer. There are many interesting analyses possible. For example, one can study the relation between the level of information in the compiler feedback and the benefit for a programmer. This effect can be on the shortterm, for finding a solution for his current programming exercise, as well as for the longterm, for the benefit of his learning process.

Fully evaluating Helium also requires other forms of empirical experimentation. For example, surveys and in vitro experiments might reveal other interesting aspects and in turn can strengthen empirical results found in the logging data sets.

The approach of analyzing Helium through loggings can also be applied to other compilers. Currently a logging facility is being implemented in GHC. Combining knowledge from multiple compilers can ultimately provide a proven model of how people (learn to) program. Establishing a widely accepted model certainly is a multi-disciplinary endeavor.

7.2 Feedback to the programmer

Presenting feedback to a programmer about his programming performance is an interesting tool in stimulating an optimal learning process. Unfortunately, we currently know little about the optimal learning curve for novice programmers. More research like discussed in the previous section, is necessary for this. However, in a class setting, we could investigate the differences between analysis results of a programmer compared to the rest of the attending students, or an experience group of programmers. The students do need to work on the same exercises. Research should answer the question, whether presenting these results to a programmer, is a proper stimulation for him, and which (combination of) analyses represent the progress of a student best.

7.3 Feedback to the lecturer

Similar as for the students, the lecturer supervising the lab session, could also be informed about the analyses results of the students. He could be informed whether the students use newly introduced programming constructs and concepts, as well as check whether separate students follow the overall trends of the complete group of students. The lecturer could use this information to improve his lectures, for instance, by focusing on problematic (and often occurring) errors.

Bibliography

[1] T. Chalmers' Haskell Interpreter and Compiler. http://www.cs.chalmers.se/~augustss/hbc/hbc.html.

[2] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[3] Free Software Foundation. Gnu project. http://www.gnu.org.

[4] S. C. Grubb. ploticus. http://ploticus.sourceforge.net.

[5] J. Hage. The helium logging facility. Technical Report UU-CS-2005-055, Institute of Information and Computing Sciences, Utrecht University, 2005.

[6] Haskell Hierarchical Libraries. http://www.haskell.org/ghc/docs/latest/html/libraries/index.html.

[7] B. Heeren and J. Hage. Type class directives. 2005.

[8] B. Heeren, J. Hage, and D. Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59 – 80, Cork, September 2003.

[9] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[10] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.

[11] R. Hinze. Fun with phantom types. In J. Gibbons and O. d. Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, Basingstoke, Hampshire, 2003.

[12] M. C. Jadud. A first look at novice compilation behavior using BlueJ. 2004.

[13] S. Joosten, K. van den Berg, and G. V. D. Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1):49–65, 1993.

[14] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, 1998.

[15] C. R. Litecky and G. B. Davis. A study of errors, error-proneness, and error diagnosis in Cobol. *Commun. ACM*, 19(1):33–38, 1976.

[16] S. Marlow and S. P. Jones. The Glasgow Haskell Compiler. http://www.haskell.org/ghc/.

[17] P. G. Moulton and M. E. Muller. Ditran - a compiler emphasizing diagnostics. *Commun. ACM*, 10(1):45–52, 1967.

[18] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs, 2006. Submitted to ICFP'06.

[19] C. Ryder and S. Thompson. Software Metrics: Measuring Haskell. In M. van Eekelen and K. Hammond, editors, *Trends in Functional Programming*, September 2005.

[20] Software Technology Website. http://www.cs.uu.nl/groups/ST.

[21] The Haskell homepage. http://www.haskell.org.

[22] The Haskell Interpreter Hugs. http://www.haskell.org/hugs/.

[23] The nhc98 compiler. http://www.cs.york.ac.uk/fp/nhc98/.

[24] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: a quantitative study. *J. Syst. Softw.*, 28(1):9–18, 1995.

[25] A. van IJzendoorn, D. Leijen, and B. Heeren. The Helium compiler. http://www.cs.uu.nl/helium.

[26] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 15–17, 2003*, pages 224–235. ACM Press, 2003.

[27] M. V. Zelkowitz. Automatic program analysis and evaluation. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 158–163. IEEE Computer Society Press, 1976.

[28] M. V. Zelkowitz and D. R. Wallace. Experimental validation in software engineering. 1997.

Appendix A

Static error codes overview

Reported error codes in the static compilation phase and referring descriptions.

unva	undefined variable
unco	undefined constructor
untc	undefined type constructor
untv	undefined type variable
unev	undefined exported module
nfts	type signature without definition
dude	duplicated definition
dutc	duplicated type constructor
duco	duplicated constructor
duts	duplicated type signature
duva	duplicated variable in pattern
am	arity mismatch for (type) constructor
da	arity mismatch for function definition
wf	filename and module name don't match
nv	pattern defines no variables
nfde	fixity declaration without definition
tv	type variable application
ls	last statement is not an expression
ts	recursive type synonym

Appendix B

Helium source compilation

All Helium compiler versions discussed in this thesis, can be compiled using GHC 6.2.2 (based on GCC, 3.4.2 20041017). When compiling the sources the following issues may occur:

- *An error on the use of the fromInt function, which is not provided any more. Occurrences can be replaced by the new fromIntegral function.*
- *Parse error in lvm/src/lib/common/Set.hs on '\' by the C-Preprocessor. Appending " – text" solves the problem.*
- *An error on a top-level splice, related to Template Haskell (in parsec/ParsecPerm.hs) $x. Solved by putting a space between $ and x.*
- *A GCC error, reporting conflicting types for 'snprintf' and 'vsnprintf' in 'lvm/src/runtime/common/misc.h' and 'lvm/src/runtime/core/misc.c'. Solved by commenting the problematic part of code, by prepending '/*' and appending '*/' on line 76 and line 92 of 'lvm/src/runtime/common/misc.h' and line 191 and line 206 in 'lvm/src/runtime/core/misc.c'*
- *When compiling the target* `coreasm`*, the version of some files may conflict, mentioning version 6022 and 6041, referring to GHC 6.2.2 and GHC 6.4.1. The makefile of some* `lvm` *parts do not use the GHC compiler variable as is done in the Helium (main) makefile, causing conflicts when another GHC is found in the search path (by for instance the use of Nix) than by the ./configure script. Providing only one version GHC via the search path solves this problem.*

Appendix C

Thesis module overview

With this thesis several tools are developed. A major component in this tool set is NEON, a statistical analysis combinator library. Furthermore, we developed also several functionalities for analyzing Helium loggings in specific. For this we also used NEON. This appendix shows an overview of all developed source modules.

C.1 Neon modules

The NEON library consists out of the following modules:

Statistics.BasicTypes.hs	Elementairy data types for the statistical library
Statistics.Completion.hs	Functions for the completion of missing values
Statistics.Ordering.hs	Functions for the ordering of values
Statistics.SummaryMeasures.hs	Functions and data types for computing statistical aggregations
Statistics.DescriptiveAnalysis.hs	Re-export of common descriptive statistics modules
Statistics.DescriptiveAnalysis.PrimAC.hs	Elementairy analysis types and combinators
Statistics.DescriptiveAnalysis.KeyableAC.hs	*Keyable* type class and *Keyable* combinators
Statistics.DescriptiveAnalysis.DescriptiveKeyAC.hs	*DescriptiveKey* type class and *DescriptiveKey* combinators
Statistics.DescriptiveAnalysis.KeyStringImpl.hs	Implementation for *Keyable* and *DescriptiveKey* for *String* as key type
Statistics.DescriptiveAnalysis.KeyMaybeAImpl.hs	Implementation for *Keyable* and *DescriptiveKey* for *Maybe a* as key type
Statistics.DescriptiveAnalysis.KeyHistoryListImpl.hs	Implementation for *Keyable* and *DescriptiveKey* for *KeyHistory* as key type
Statistics.Presentation.hs	Re-export of common presentation modules
Statistics.Presentation.TextRepresentation.hs	Functions for rendering tables
Statistics.Presentation.HtmlRepresentation.hs	Functions for rendering `HTML` output
Statistics.Presentation.MarkUpDoc.hs	Functions for combining and rendering *MarkUpDoc*
Statistics.Presentation.Ploticus.hs	Functions for rendering `ploticus` scripts
Statistics.Presentation.RepresentationActions.hs	Common presentation (IO) actions

C.2 Modules developed for Helium analysis

Thesis.Main.hs	Main program module
Thesis.Args.hs	Functions for handling the Helium analysis tool parameters
Thesis.Settings.hs	Setting for the Helium analysis tool
Thesis.Select.hs	Functions for selecting values
Thesis.Metrics.hs	Elementairy Helium metrics
Thesis.AnalysisPredefined.hs	Definition of predefined Helium case study analyses
Thesis.Helium.HeliumInfo.hs	Defined Helium information
Thesis.Helium.Logging.hs	Definition of Helium logging and related data types
Thesis.Helium.LogfileParser.hs	Parser of logfile of Helium loggings
Thesis.Helium.LoggingPP.hs	Logging pretty printer
Thesis.Helium.Parser.Lexer.hs	Lexer of Helium sourcecode
Thesis.Helium.Parser.LexerMessage.hs	Error reporting of the Helium lexer
Thesis.Helium.Parser.LexerMonad.hs	Lexer monad of Helium sourcecode
Thesis.Helium.Parser.LexerToken.hs	Lexical tokenizer of Helium
Thesis.Helium.Parser.Utils.hs	Auxiliary utilities for the Helium lexer
Thesis.Helium.Parser.ResearchLexer.hs	Custom developed lexer for lines of code, comments, and empty lines research
Thesis.Research.ModuleLengthResearch.hs	Analysis research studying module lengths
Thesis.Research.PhaseResearch.hs	Analysis research studying logged phases
Thesis.Research.LineBasedInterpretationResearch.hs	Analysis research studying source lines
Thesis.Research.CompilationDistribution.hs	Analysis research studying compilation intervals
Thesis.Research.HintRateResearch.hs	Analysis research studying hint ratios
Thesis.Research.TypingTillCodeResearch.hs	Analysis research studying type error repair
Thesis.Research.DataSetActivityResearch.hs	Analysis research studying logging sets
Thesis.Research.AbstractIntMetricResearch.hs	Abstract research used by other researches
Thesis.Research.CheckLogDate.hs	Analysis research studying the parsing of the log date
Thesis.Research.Common.hs	Common analyses, used by other researches
Thesis.Research.ThesisACExample2.hs	First combinator example presented in thesis
Thesis.Research.ThesisACExample.hs	Second combinator example presented in thesis
Thesis.Grouping.BasicTypes.hs	Elementairy types for grouping
Thesis.Grouping.HeliumErrors.hs	Functions for grouping Helium errors
Thesis.Grouping.LexerToken.hs	Functions for grouping lexer tokens errors
Thesis.Grouping.Logging.hs	Functions for grouping loggings errors
Thesis.Grouping.Similar.hs	Functions for grouping based on similarities

C.3 Modules used by both projects

The following modules were used in both projects.

Auxiliary.hs	Auxiliary functions
CommonIO.hs	Auxiliary functions for common IO tasks
Grouping.hs	Auxiliary grouping functions
Distribution.Compat.FilePath.hs	Common file path handling functions (from Haskell's Hierarchical Libraries)

Appendix D

Statistical analysis combinator overview

The NEON library provides three sets of primitive analysis combinators, each having a different level of specialization.

D.1 Basic primitive analysis combinators

The following primitives are defined in the module *Statistics.DescriptiveAnalysis.PrimAC*.

Primitives for computing a value from an input.

$basicAnalysis''' :: (a \to b \to keya \to keyb) \to (a \to b) \to Analysis\ keya\ a\ keyb\ b$
$basicAnalysis'' :: (a \to keya \to keyb) \to (a \to b) \to Analysis\ keya\ a\ keyb\ b$
$basicAnalysis' :: (keya \to keyb) \to (a \to b) \to Analysis\ keya\ a\ keyb\ b$
$basicAnalysis\_ :: keyb \to (a \to b) \to Analysis\ keya\ a\ keyb\ b$

The primitive presented in this thesis.

$basicAnalysis :: (keya \to keyb) \to (a \to b) \to Analysis\ keya\ a\ keyb\ b$
$basicAnalysis = basicAnalysis'$

Primitives for computing an aggregate value from the complete analysis input.

$aggregateGrpsAnalysis''' :: ([a] \to b \to [keya] \to keyb) \to ([a] \to b) \to Analysis\ keya$
$aggregateGrpsAnalysis'' :: ([a] \to [keya] \to keyb) \to ([a] \to b) \to Analysis\ keya\ a$
$aggregateGrpsAnalysis' :: ([keya] \to keyb) \to ([a] \to b) \to Analysis\ keya\ a\ keyb\ b$

The primitive presented in this thesis.

$aggregateAnalysis :: ([keya] \to keyb) \to ([a] \to b) \to Analysis\ keya\ a\ keyb\ b$
$aggregateAnalysis = aggregateGrpsAnalysis'$

Primitives for grouping subjects.

$groupAnalysis'' :: ([[a]] \to keya \to [keyb]) \to ([a] \to [[a]]) \to Analysis\ keya\ [a]\ keyb\ [a]$
$groupAnalysis' :: ([a] \to keya \to keyb) \to ([a] \to [[a]]) \to Analysis\ keya\ [a]\ keyb\ [a]$
$groupAnalysis'\_ :: (a \to keya \to keyb) \to ([a] \to [[a]]) \to Analysis\ keya\ [a]\ keyb\ [a]$

The primitive presented in this thesis.

$groupAnalysis :: (a \to keya \to keyb) \to ([a] \to [[a]]) \to Analysis\ keya\ [a]\ keyb\ [a]$
$groupAnalysis = groupAnalysis'\_$

Primitives for selecting subjects.

$selectGrpAnalysis'' :: (a \rightarrow key1 \rightarrow key2) \rightarrow (a \rightarrow Bool) \rightarrow Analysis\ key1\ a\ key2\ a$
$selectGrpAnalysis' :: (key1 \rightarrow key2) \rightarrow (a \rightarrow Bool) \rightarrow Analysis\ key1\ a\ key2\ a$

The primitive presented in this thesis.

$selectGrpAnalysis :: (key1 \rightarrow key2) \rightarrow (a \rightarrow Bool) \rightarrow Analysis\ key1\ a\ key2\ a$
$selectGrpAnalysis = selectGrpAnalysis'$

D.2 Keyable primitive analysis combinators

The following primitives are defined in the module $Statistics.DescriptiveAnalysis.KeyableAC$.

Primitives for computing a value from an input.

$basicAnalysis'' :: Keyable\ key \Rightarrow (a \rightarrow b \rightarrow key) \rightarrow (a \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$basicAnalysis' :: Keyable\ key \Rightarrow (a \rightarrow key) \rightarrow (a \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$basicAnalysis\_ :: Keyable\ key \Rightarrow key \rightarrow (a \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$

The primitive presented in this thesis.

$basicAnalysis :: Keyable\ key \Rightarrow key \rightarrow (a \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$basicAnalysis = basicAnalysis\_$

Primitives for computing an aggregate value from the complete analysis input.

$aggregateGrpsAnalysis'' :: Keyable\ key \Rightarrow ([a] \rightarrow b \rightarrow key) \rightarrow ([a] \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$aggregateGrpsAnalysis' :: Keyable\ key \Rightarrow ([a] \rightarrow key) \rightarrow ([a] \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$aggregateGrpsAnalysis\_ :: Keyable\ key \Rightarrow key \rightarrow ([a] \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$

The primitive presented in this thesis.

$aggregateGrpsAnalysis :: ([keya] \rightarrow keyb) \rightarrow ([a] \rightarrow b) \rightarrow Analysis\ keya\ a\ keyb\ b$
$aggregateGrpsAnalysis = aggregateGrpsAnalysis'$

Primitives for grouping subjects.

$groupAnalysis'' :: Keyable\ key \Rightarrow ([[a]] \rightarrow [key]) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$
$groupAnalysis' :: Keyable\ key \Rightarrow ([a] \rightarrow key) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$
$groupAnalysis\_ :: Keyable\ key \Rightarrow (a \rightarrow key) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$

The primitive presented in this thesis.

$groupAnalysis :: Keyable\ key \Rightarrow (a \rightarrow key) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$
$groupAnalysis = groupAnalysis\_$

Primitives for selecting subjects.

$selectGrpAnalysis'' :: Keyable\ key \Rightarrow (a \rightarrow key) \rightarrow (a \rightarrow Bool) \rightarrow AnalysisFK\ key\ a\ a$
$selectGrpAnalysis' :: Keyable\ key \Rightarrow key \rightarrow (a \rightarrow Bool) \rightarrow AnalysisFK\ key\ a\ a$

The primitive presented in this thesis.

$selectGrpAnalysis :: Keyable\ key \Rightarrow key \rightarrow (a \rightarrow Bool) \rightarrow AnalysisFK\ key\ a\ a$
$selectGrpAnalysis = selectGrpAnalysis'$

D.3 DescriptiveKey primitive analysis combinators

The following primitives are defined in the module $Statistics.DescriptiveAnalysis.DescriptiveKeyAC$.

Primitives for computing a value from an input.

$basicAnalysis' :: DescriptiveKey\ key \Rightarrow String \rightarrow (a \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$

$basicAnalysis\_ :: DescriptiveKey\ key \Rightarrow (a \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$

The primitive presented in this thesis.

$basicAnalysis :: DescriptiveKey\ key \Rightarrow String \rightarrow (a \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$basicAnalysis = basicAnalysis'$

Primitives for computing an aggregate value from the complete analysis input.

$aggregateGrpsAnalysis'' :: DescriptiveKey\ key \Rightarrow String \rightarrow ([a] \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$aggregateGrpsAnalysis' :: DescriptiveKey\ key \Rightarrow ([a] \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$

The primitive presented in this thesis.

$aggregateGrpsAnalysis :: DescriptiveKey\ key \Rightarrow String \rightarrow ([a] \rightarrow b) \rightarrow AnalysisFK\ key\ a\ b$
$aggregateGrpsAnalysis = aggregateAnalysis''$

Primitives for grouping subjects using the *DataInfo* class.

$groupAnalysis'' :: (DescriptiveKey\ key, DataInfo\ b) \Rightarrow ([[a]] \rightarrow [b]) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$
$groupAnalysis' :: (DescriptiveKey\ key, DataInfo\ b) \Rightarrow ([a] \rightarrow b) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$

Primitives for grouping subjects using two strings to describe the group index, instead of the *DataInfo* class.

$groupAnalysis''\_ :: DescriptiveKey\ key \Rightarrow ([[a]] \rightarrow [(String, String)]) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$
$groupAnalysis'\_ :: DescriptiveKey\ key \Rightarrow ([a] \rightarrow (String, String)) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$
$groupAnalysis\_ :: DescriptiveKey\ key \Rightarrow (a \rightarrow (String, String)) \rightarrow ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$

The primitive presented in this thesis.

$groupAnalysis :: (DescriptiveKey\ key, DataInfo\ b) \Rightarrow (a \rightarrow b) \rightarrow$
$\qquad ([a] \rightarrow [[a]]) \rightarrow AnalysisFK\ key\ [a]\ [a]$
$groupAnalysis\ keydescr\ grpfun = groupAnalysis'\ (keydescr\ .\ head)\ grpfun$

Primitives for selecting subjects.

$selectGrpAnalysis''\_ :: DescriptiveKey\ key \Rightarrow String \rightarrow (a \rightarrow Bool) \rightarrow AnalysisFK\ key\ a\ a$
$selectGrpAnalysis'\_ :: DescriptiveKey\ key \Rightarrow (a \rightarrow Bool) \rightarrow AnalysisFK\ key\ a\ a$

The primitive presented in this thesis.

$selectGrpAnalysis :: DescriptiveKey\ key \Rightarrow String \rightarrow (a \rightarrow Bool) \rightarrow AnalysisFK\ key\ a\ a$
$selectGrpAnalysis = selectGrpAnalysis''\_$

Appendix E

Program logging functionalities

As part of this thesis project, we provide a set of data types and functionalities for handling and evaluating Helium program loggings. This chapter discusses these auxiliary functions.

E.1 Data set logfile parser

The module *Thesis.Helium.LogfileParser* provides a parser to retrieve the loggings from a logfile, provided with each logging data set. This logfile is stored in the base directory of a logging data set and provides the basic information of each logging event, including a reference (file path) to the logged program source. The function $parseLogfile :: FilePath \rightarrow IO\ [Logging]$ can be used to parse the complete set of loggings from the logfile, and returns these loggings in a list of type $[Logging]$.

E.2 The *Logging* data type and related functionalities

The module *Thesis.Helium.Logging* defines a Helium compiler logging by the *Logging* data type. This data type encapsulates the basic attributes of a logging, like the username of the student, the compilation phase in which the compilation process ended, and various other attributes. This module also provides a range of additional operations on the *Logging* data type.

The *Logging* data type is defined as:

$$\begin{aligned}&\textbf{data}\ Logging = Log\{\ username\ &&:: Username,\\ &\qquad\qquad\quad\ \ phase\ &&:: Phase,\\ &\qquad\qquad\quad\ \ heliumVersion\ &&:: HeliumVersion,\\ &\qquad\qquad\quad\ \ logPath\ &&:: Path,\\ &\qquad\qquad\quad\ \ logDate\ &&:: CalendarTime,\\ &\qquad\qquad\quad\ \ staticErrorCodes\ &&:: [StaticErrorId]\\ &\qquad\qquad\ \}\\ &\ \ \textbf{deriving}\ (Show, Eq, Read)\end{aligned}$$

The data type is designed using record notation and provides a set of fieldlabels to access the basic information of the loggings.

The following fieldlabels are available: *username* contains the username of the logged programmer, *phase* is the phase in which the compilation process ended, *heliumVersion* contains the version which was used by the student to compile, *logPath* id the file path referring to the source program of the logging, *logDate* is the date and time (a timestamp) of the logging, and *staticErrorCodes* are the static error codes, which were found in the static analysis phase of the compilation.

The (data) types used by the *Logging* data type are:

$$\begin{aligned}&\textbf{newtype}\ Username = Username\ String\\ &\ \ \textbf{deriving}\ (Eq, Show, Read, Ord)\end{aligned}$$

```
data Phase = Lexical | Parsing | ResolOp | Static | Typing | CodeGen | InternalErr
  deriving (Show, Eq, Read, Ord, Enum)

data HeliumVersion = HeliumVersion{
                       heliumVersionNumber :: String,
                       buildDateRef :: String
                     }
  deriving (Eq, Read, Ord, Show)

data Path = Path{
              baseDir :: String,
              userDir :: String,
              loggingDir :: String,
              sourceFile :: String
            }
  deriving (Eq, Read, Show)

type StaticErrorId = String
```

In addition to the fieldlabels, the module *Thesis.Helium.Logging* also provides a set of helpful functions. To retrieve the source code, compiler output, and the verbose compiler output of a logging, one can use:

```
sourcecode            :: Logging → String
compileroutput        :: Logging → String
compileroutputverbose :: Logging → String
```

These functions read files from disk in a strict manner. Reading a file is typically an IO-action, but for practical reasons we use *unsafePerformIO* to leave the *IO monad*. Using this approach, allows us to have similar types for the logging attributes (the fieldlabels of the *Logging* data type), as well as for the source and the compiler output of a logging. We can justify this approach, because we consider the data sets to be immutable. An analysis does not change any data from our data sets. The files are also write-protected on a file level. So we safely can assume that reading a file, related to a logging is side-effect free. During the execution time of an analysis program, the content is not supposed to change. As such in our situation, reading a file is very similar as accessing a database.

An additional helper function is *unparsedLogDate*. This function returns the unparsed logging timestamp as string. In some cases an analysis can run faster without parsing the timestamp. The timestamp is actually a part of the filepath, referring to the location of the logged program file.

```
unparsedLogDate :: Logging → String
unparsedLogDate = loggingDir . logPath
```