Analyzing errors in Helium loggings

Mathijs Swint

MSc Thesis

August 7, 2010

INF/SCR-09-74

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Supervisor:
dr. Jurriaan Hage

Contents

Chapter 1

Introduction

1.1 Context

In this thesis we gather metrics about the programming behaviour of students using Haskell [P+03]. We achieve this by analyzing the data provided by the Functional programming course at Utrecht University. The functional programming course uses the Helium compiler [HLvI03], which has a built in logging facility. This facility has contributed a large part of the data on which our work is based.

Helium is both a compiler and a programming language. The Helium compiler compiles the Helium language, a (simplified) subset of Haskell. The compiler focuses on generating good error messages, which may indicate how students should adapt their code to compile successfully. It also gives a large number of hints which indicate code smells and a violation of code conventions in compiled programming code, for example indicating when an "otherwise" is missing from a case statement. The Helium compiler is supplied with a logging facility, which stores data about the compiled source files.

The Neon project is a project for extracting data from Helium loggings. It was started by Van Keeken and Hage [HK07]. The project has resulted in a tool set which can be used to study the data provided by the Helium loggings. The tool set allows users to write analyses which are used to data mine the Helium loggings, providing metrics about the data stored in these loggings, in an embedded domain specific language (EDSL). It also provides functions for graphically presenting the results of these mining operations.

An EDSL is a specific form of a domain specific language (DSL), a program-

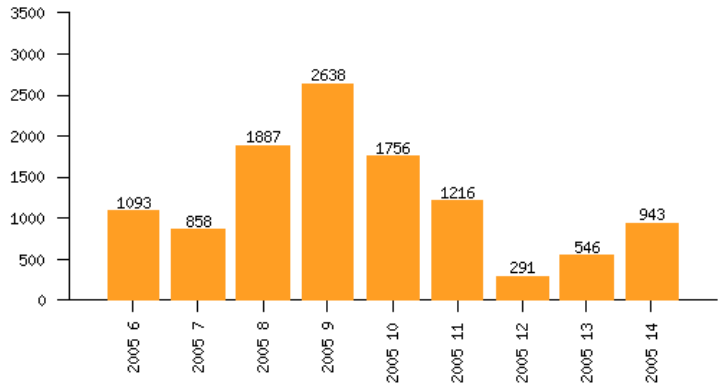

Figure 1.1: Total amount of loggings per week

ming language dedicated to a particular problem. DSLs provide a syntax which is more convenient and generally more compact than a generic language can offer for the same domain. An EDSL is a language which is provided as a library for an existing general purpose language, dedicated to a specific problem. The general purpose language which Neon is built into is Haskell.

Neon provides a combinator library for writing analyses on the Helium Loggings. Combinators are a familiar concept to many Haskell programmers. They are, for example, frequently used when writing parsers. Combinators are functions which take multiple domain specific elements as their input and provide a new function within the same domain. In our parser example they are used to create parsers from other parsers. Our combinator library is used to create analyses which are used to generate our results. This allows us to create complex analyses out of simpler ones.

A very simple example of such an analysis is to compute the number of compilations per week. This can be written as follows:

```
totalLoggingsPerWeek :: AnalysisKH [Logging] Int
totalLoggingsPerWeek = basicAnalysis "number of loggings" length
    <o> groupPerWeek
```

Here, the <o> operator is an example of a combinator, one that implements analysis composition.

Neon can render the results from our example as a bar chart image, as seen in figure 1.1.

1.2 Research Question

Neon can be used to study data in the loggings supplied by the Helium compiler. We are able to like answer questions like "How often do students compile programs?", "What is the average size of a program?" or "How long does it take students to change a program which does not compile into one which does?". It is however not able to help us answer questions about how students threat the errors and warnings presented to them. These kind of questions are interesting to anyone studying the effects of the course.

In this thesis we investigate how to support answering questions about the errors and warnings presented to users by the Helium compiler, by means of the Neon tool set. For example, can we use the data logged by the Helium compiler to develop a view of how students program? Can the data retrieved from the loggings answer questions like: "How do students handle warnings and errors?" Our method for answering these questions will be to develop a number of analyses which illustrate how one should approach such studies, which were not possible using the original Neon framework.

1.3 Contributions

Originally, Neon worked as a fully compiled executable, with a number of built-in analyses. This meant developing (and testing) a new analysis required us to write an analysis, recompile the source code into an executable and test our analysis on a dataset. We have dramatically shortened this development iteration when building new analyses. Helium has been made faster, allowing us to develop new analyses in an interactive `ghci` session. This has the benefit of being able to keep parsed data files in scope: a large part of Neon's usual running time is spent parsing the Helium logs and related files. The Haskell interpreter, `ghci`, can keep these variables in scope, meaning they do not have to be re-parsed when either performing a new analysis or altering an existing one. These adaptations improve the tool set, allowing us (and others) to develop analyses faster than before.

Our addition of course and student data to Neon allows us to target a specific kind of students in our analyses, which allows our metrics to be more concise. Course data contains such information as what grade a student received for the course and what grades he / she received for practical exercises. Student data contains information about the nature of students, that cannot be extracted from the Helium loggings.

Examples of student data are, as part of which bachelor program a student

followed the functional programming course and the total amount of students who enrolled in the course. This information allows us to be concise in our analyses, by identifying (un-)desired data in our set of loggings. For example, if we would like to examine how students learn to develop programs it can be useful to exclude those students who have previously followed the functional programming course, since they have previous knowledge of the programming language being taught.

While Helium stores an exact copy of the source files compiled by a user, Neon was unable to study the content of actual errors which resulted from compiling that file. It was only able to retrieve the phase in which compilation failed, which is stored in the regular log files. We extend Neon to be able to investigate the actual errors and warnings. We have created a parser which is able to interpret the output which is presented to the Helium users at compile time. This allows us to perform studies based on the actual errors and warnings messages shown to the users.

We use the improved Neon system to perform a number of interesting analyses on the Helium loggings. We show the amount of parse errors students make and what kind of parse errors are most frequently encountered. We disclose which warnings students encounter and how students handle them. The metrics we disclose can give insight into which future studies into the way people program may be useful. These analyses study how students handle parse errors and warnings.

During our work we encountered a number of recurring patterns in analysis design. These resulted in a number of new combinators and a number of analyses. One of our new combinators allows us to easily present data over multiple figures. The analyses can be reused when developing new analyses. In our analyses we illustrate what indicates a complete sequence of loggings, meaning that no compiles took place which we did not log. We also show which loggings are likely to belong to the same programming session, indicating two sequential compiles took place while students did not engage in other compiles between two compilations. We use this analysis when checking how long it takes for students to resolve warnings. We give indications of their validity and indicate parameters for tuning these analyses.

1.4 Outline

We start out by discussing the state of the Neon framework in chapter 2. We consider the data available to us and how we made that data available in Neon in chapter 3. In chapter 4 we introduce the notion of cross-cutting

concerns (recurring patterns) and show the cross-cutting concerns we have addressed. Overall, in chapters 5, 6, 7, 8 we discuss the analyses we performed on the dataset of the functional programming course of 2005. In chapter 5 we illustrate frequently encountered errors and warnings. In chapter 6 we watch how warnings are fixed over time. In chapter 7 we gather information about which parse errors are frequently encountered. In chapter 8 we look at whether students with high grades make fewer errors. Our conclusions can be found in chapter 10. In appendix A we describe and illustrate the code of a large number of extensions we added to Neon.

Chapter 2

Context

This thesis project builds on research previously done at the Software Technology group at the Utrecht University. It builds on the work done in the context of the TOP research program, primarily on the research and works by Van Keeken on the Neon Program Analysis Tool. Neon is a tool for analyzing the loggings produced by the Helium compiler, developed at the Utrecht University.

2.1 Helium

Helium [Haga] is a Haskell dialect developed at the Utrecht University as a tool to show the validity of research done within the TOP project [Hagb]. It has been successfully employed in a number of functional programming courses since 2002. It was developed with one specific purpose in mind: The teaching of Haskell to novice programmers. For this reason a logging facility was built into the Helium compiler. In essence it copies every source file compiled by students and stores them on a central server. Neon aids us in performing analyses on this dataset.

The Helium compiler focuses on generating 'better', more comprehensible, error messages. Early incarnations of Haskell compilers failed at showing error messages which are understandable to novice students. [HLvI03] It is important that novice programmers understand error messages, since this may directly impact their interest in the new programming language. To aid novice programmers Helium attempts to present plain and simple error messages and locating the exact position of the error, by both providing the line and column of the error. Not only does Helium attempt to provide a very clear error message, it also attempts to offer a hint on how to solve the error.

To allow studying the effect of these improved error messages one needs to perform empirical research on the results of these compilers. This is where the logging facilities offered by Helium come in. The Helium logger stores a large number of properties of the compiled files. Amongst the stored data is:

- An identification of the programmer
- The time of compilation
- The exact version of Helium used to compile the source
- In recent years, the exact parameters passed to Helium
- The compilation phase in which the compiler failed
- The complete source of the compiled file and included modules in a separate file

This collected data allows us to do the required empirical studies to establish the usefulness of the Helium compiler, without directly relying on user experience, which is more common in empirical studies [HK07].

2.2 Neon

Neon has been developed as a tool to perform analysis on the large amounts of data described as produced by the Helium logging facility. It is a tool designed to do empirical studies on the collected loggings. It was produced by Van Keeken to aid him in analyzing neon's loggings for his master thesis.

According to Van Keeken and others [LHPT94; ZW97], empirical studies are rarely used when attempting to back up claims concerning software tools. Arguments to support claims are generally backed up by arguments resulting from logical reasoning. This is a pity, since empirical studies can be a powerful tool when backing up ones claims. Empirical studies often involve a large amount of manual labour, especially while collecting and analyzing the data to perform these studies. Neon aids in removing a large amount of the tedious manual work when performing this type of research on the Helium compiler.

Neon works by interpreting the loggings created by Helium. Van Keeken created a number of analytical combinators, allowing for the easy composition of a number of simple analytical operations. Neon should be capable of determining wheter or not loggings should be grouped on different properties. This allows Neon to identify both individuals and groups of

individuals working together, which compiles belong to the same program and even which compiles belong to the same exercise submission for an individual course.

Van Keeken provided a framework for metric calculating operations by building Neon, in the form of the combinator based analysis engine he developed, which he named statistical combinators. These combinators provide the basis for any analysis performed by Neon. Combinators are a familiar concept to most Haskell programmers, offering an easy way to combine functions. In essence a combinator allows us to take one or more functions of the *Analysis* type and transform these into anther function of the *Analysis* type. A popular application of combinators is the parser combinator library provided by Daan Leijen [LM01].

The combinator library is built up from a number of basic combinators (each with a different function). There are combinators which can produce groupings, create compositions (<∘>), perform function application (<$>), split the flow of the analysis to produce seperate outputs (<&>) and many more. These can be used to create more advanced analyses.

The results of Neon's analyses are resented in graphical form. A number of statistics can be rendered into either graphic or table based form to describe the collected statistics. The graphics are created using a third party tool, ploticus [Gru]. The graphical outputs can be combined into a descriptive report, either in an html or LaTeX format. A number of representations for different types of analyses has been provided, for example: An analysis describing the average amount of lines of code per week in a given set of logs can be displayed using a table or a bar plot.

The combination of this work resulted in Van Keeken being able to create a number of analyses. He formed a number of hypotheses which he intended to (dis-)prove using the analysis results provided by neon. This resulted in Van Keeken developing the following analyses:

- An analysis which calculates the average module length per week.
- An analysis to calculate the number of loggings per phase per week.
- An analysis which compares the number of the code / comments / empty lines per week.
- An analysis which compares the compilation interval per week.
- An analysis which compares the average time it takes to repair an error per week.
- An analysis which compares the number of error messages which contain hints per week.

2.3 Related work

Similar studies have been done on other programming languages. An example of these are the analyses done by Jadud using BlueJ at the University of Kent [Jad05] that closely resemble those that can be performed by Neon. BlueJ is an IDE capable of logging the compiles of students. Like Helium it stores some meta-data and is capable of logging complete sources to a central server [Kol]. He studied the results of the works of 63 students and analysed the loggings gained by monitoring these students. Some examples of the analyses performed by Jadud are "Error types and distribution" and "Time between compilation events".

Chapter 3

Neon Data

3.1 Considerations

3.1.1 Available Data

We will have to deal with a large amount of data provided by the Helium logger. These can be processed by Neon as illustrated in section 2.2. To answer our research questions we focus on a large dataset, namely the one provided by the functional programming course in 2004/2005. We have managed to gather a large amount of statistics to supplement the data in the original log files. This allows us to compare a number of new statistics not available to Van Keeken when he did his research. We have managed to determine the grades received while following the course on functional programming. We will attempt to couple this data this student data to the individual loggings. This will allow us to compare the grades a student received to "quality" of his submitted work. Some examples of quality measurements would be "time taken to solve an error" or "number of errors (or warnings) per compiled source". This will allow us to study if what we can show a relationship between these criteria, which are generally seen as indications of a students understanding of the course, and the students grade.

3.1.2 Helium Versions

A large number of different Helium versions have been used over the previous years in which the Functional Programming course was thought at the University of Utrecht. A new version has been released almost every year. This means that there are many Helium versions to take into account.

Van Keeken did attempt taking different Helium versions into account, but his work work failed to materialise.

Although he failed at taking different Helium versions into account, even the logging output of Helium versions differs. For example early versions of the Helium logging facility only included the actual source file being compiled by the compiler, not the included version Modules. Since these files had to be compiled at some point it is possible to recover this data. Old loggings are currently reconstructed to the latest format by a number of Perl scripts, developed by Jurriaan Hage. We will attempt not to modify the input preparation to allow us to keep using these scripts to generalise the logging files. It seems unlikely that the data in the current logging format will be insufficient for our purposes.

Tempting as it may be, it seems unlikely we will be able to treat all source files as if they were used with one and the same Helium version. The internals of the current Helium compiler differ too much from the version originally introduced in 1999.

For some versions of Helium the output of the compiler may differ. Error messages may have been improved or changed over time to result in a better understanding or have been made more concise. This means it is unlikely we will be able to use the output of different Helium versions to perform analyses on parse errors.

We expect these versions to execute differently on source files. Some may accept new features which have not been implemented in new versions, which leads to errors in the older versions while being perfectly legitimate in a newer version. We need to make sure if we can focus on compiling sources with the latest version of Helium or if we need to implement some form of representation which allows us to make distinctions between different kinds of parse errors.

3.1.3 Cross-cutting Concerns

Doing analysis on a large dataset can be done in an ad-hoc manner. It is easy to parse the Helium log files by combining a number of tools in a standard Unix toolkit. One can easily define a number of metrics using regular expressions and line counting, which enables us to generate a number of metrics. This would allow us to do things like counting the number of successful compiles per student, by piping a number of tools.

It is however very hard to do in depth analysis which require a less linear approach to develop. Imagine compiling the submitted results to determine the actual error messages or compilation result as presented to students then

analyzing them to generate some statistics about the actions performed by a student to fix the error. Neon was developed to address these problems, allowing users to build upon and combine a number of "basic" operations and combine these to an interesting analysis which can show us some interesting metrics.

3.1.4 Reusing Helium

Currently Neon is only able to perform analysis on static statistics collected from a number of simple loggings, without depending on the actual contents of the program. We would like to extend Neon to make it possible to perform analysis on the actual contents of the stored Helium source files. We expect to encounter a number of difficulties while trying to achieve this extension, as described in the following subsections.

In the analyses performed by Van Keeken only a few parts of Helium were reused, namely parts of the Lexer. Reusing the lexer allowed Van Keeken to do analysis on the Token level of Helium's compilation process. Developing a Helium lexer is critical to answer a number of interesting questions, such as the ones posed by Van Keeken. One of his analyses focused on comparing the number of lines of comment to actual lines of code. One could solve this problem by writing a regular expression which focuses on determining which sections of a source file are comments, but a far more reliable way is to reuse the lexer. Reusing the lexer allows no discrepancies between the analysis and the actual compiler results presented to students using Helium.

To perform our research we would have to include the parser provided by the Helium project. Unfortunately it is doubtful we will be able to build on the lexer included by Van Keeken, since his work on the lexer only includes counting certain tokens. We will need to devise some way of using the Helium parser which enables us to differentiate between different types of parse errors. We have considered several different approaches, which are listed below.

3.1.5 Analysing Logged Errors

Output Analysis A first attempt at doing some analysis on parse errors will be conducted by simply calling Helium from the command line, on the sources provided by the log files. We might be able to distinguish between different parse errors by parsing Helium's output messages. This quick fix may be enough to do a number of interesting analysis.

Parser Reuse It is not unlikely that output analysis on the Helium parser is incapable of detecting subtle differences between different types of parse errors, since they were not developed for mechanical analysis, but for easy reading. An in depth approach to developing Helium further may be to reuse the parser error from Helium to detect the exact type of failures. This approach will surely allow us to do virtually any analysis desirable on the submitted sources, but it poses a whole new set issues. First of all we need to determine how easy (or hard) it is to extract the Parser from Helium without doing any substantial modifications to its source and without having to include Helium as whole. Second of all we will have to take into account the different versions of the Helium parser developed over the years. An in depth analysis of the problems posed by this can be found in section 3.1.2.

We eventually chose to stick with output analysis. This approach offered us enough information for studying the data from the 2005 functional programming course. As section 3.1.6 explains, we only gathered data from a single year, we therefore did not need to adopt the parser for large helium changes. Besides that, the loggings from 2004 followed a similar scheme as the ones provided by the 2005 course. We have not verified our method against other years.

3.1.6 Using student data

To use student data we need to gather a large amount of data from the systems which are available at the Utrecht University. Student data is data which is not directly related to the compilation behaviour of students. It is not logged by the Helium logger. An example of student data are a students personalia. Another example would be the grades a student received for the course.

Unfortunately not all of this data is centrally stored in a location which is accessible to us. Individual teachers are obliged to keep the data on their courses for several years, allowing us to approach them with data requests. Student assistants may be able to provide data on data related to practical sessions, since they are required to grade the practical work.

3.2 Gathered data

3.2.1 Gathering student data

A large amount of properties of a student are known by our department. Many of these may be interesting to inspect in our inquiries into student behaviour. For example knowing whether or not a student took the functional programming course in which Helium was used in the previous year, as he did in the year we are investigating. This kind of data allows us to filter out disturbing factors like their knowledge of language constructs which weren't introduced yet in the current course. This type of knowledge allows to take a look at only new Haskell programmers. We have however also gathered a large amount of other data. We will describe each of them briefly.

Properties

Student type In the curriculum of students attending the 2005 course functional programming is a mandatory course for two bachelor programs at the Utrecht University, namely the computer science and cognitive artificial intelligence program. It is however an optional part of many bachelor programs and serves as an optional course for the computer science minor. The course is somewhat popular with math and physics students.

Participation This property says whether a student actually submitted any data for grading during the course. Not being an actual participant can mean a student wasn't even part of the course or dropped out. It does not tell us anything about a students actual enrolment status.

Attempt Quite a few students fail the functional programming course each year. Many of them will try to pass the course again the following year. We have managed to determine how many times a student previously enrolled in the course (for most students).

Imperative programming Haskell typically is not the first programming language students encounter at the Utrecht University. The regular curriculum of the computer science program includes an imperative programming course before the functional programming course. However, a number of students fail to pass this course. The students from other programs are not

required to follow the imperative programming course. Cognitive artificial intelligence are required to attend a course in which they are taught Prolog. We have not been able to record the grades of this course. We have recorded whether they passed or failed the imperative programming course, with which grade and whether they ever took part in the course.

Final grade Students naturally receive a grade when finishing the course. A students final grade is a combination of three grades handed out for submitted assignments and the result of a tentative exam.

Practical assignments During the course of the functional programming course students are required to hand in three programs as a practical assignment. We have gathered both the marks and the deadlines for these practical assignments.

Sex We store the gender of each student.

Enrolment status The central student administration maintains a record of each students enrolment status. For a large portion of the students we have been able to determine whether they finished their bachelor program, enrolled in a masters program and / or dropped out of their respective program.

3.2.2 Process

Neon relies on a large number of pre-processed loggings. Our pre-processed loggings were anonymized to allow us to gather data without prejudice and to prevent us from presenting private information outside our institute. There are two unique identifiers of a student within our institute, the studentid and a unique user name which is used to log on to Windows and Unix machines. The loggings recorded a students user name, which was removed during the pre-processing step. The only way for us to couple data however, is this exact user name. We therefore re-ran the pre-processing steps on our data to be able to couple our gathered student data to a student. The required data for each student was collected by dr. Jurriaan Hage.

The student data is stored in a spreadsheet, as a tab separated file, and stored with the processed sources. This allows Neon to easily parse the file. The data from the spreadsheet is parsed and coupled to each logging when

```
==================================================
= Helium Compiler 1.5-Thu-Jan-27-15_37_27-RST-2005
==================================================
Compiling group71/2005-01-07@15_38_24_960/Hello.hs
 Lexing...
 Parsing...
 Importing...
 Resolving operators...
 Static checking...
 (1,7): Warning: Variable "n" is not used
 Type inference directives...
 Type inferencing...
 (1,1): Warning: Missing type signature: hello :: a -> [[b]] -> [b]
 Desugaring...
 Code generation...
 Compilation successful with 2 warnings
```

Figure 3.1: Verbose compiler output

Neon is run on a given set of loggings, such that a students data is available for each log entry created by the student.

3.2.3 Gathering compiler output

The helium logging facility not only stores data about each compile, but also the original sources that were compiled. Van Keeken managed to recompile all sources originally submitted to the Helium system.

The regular Neon output was recompiled with the *–verbose* option. This allows the compile to tell us in which phase it threw errors or warnings and which phase compilation ended.

The verbose option creates a very well structured output, which provides a unique opportunity to re-parse its output without getting into to much trouble. An error clearly states on which line it occurs, what type of message it is throwing and the actual error message. This results in output looking like shown in figure 3.1.

A compile can trigger the compilation of imported source files. The included source files for the imports statements are compiled one by one and their result is printed. We store errors as a 3-tuple of compiler location, error type and error message, as seen in figure 3.2. This structure is easily traversable with well known Prelude functions.

type *CompilerOut* = (*Version*, [*CompiledFile*])
type *CompiledFile* = (*File*, [*CompiledPhase*])
type *CompiledPhase* = (*Phase*, [*Error*])
type *Version* = *String*
type *Error* = (*Position*, *String*, *String*)
type *Position* = (*Int*, *Int*)
type *File* = *String*

Figure 3.2: Storing compiler output

Neon's new parsing extension allows us to take a look at errors and warnings from 2004- onwards. Helium versions before this year did not contain the options to make Helium's output verbose enough to make the output presented by Helium parseable. This still presents us with 4 years from which we can analyse data. We have chosen however to limit ourselves to the data collected in the Functional programming courses in 2005.

Chapter 4

Cross-Cutting Concerns

4.1 Introduction

Cross-cutting concerns are those issues which are encountered in a large amount of analyses. In this chapter we introduce a number of new cross-cutting concerns which we have added to the helium framework.

We start by looking at two grouping analyses which we have developed. These allow us to group data by a certain predicate. The two analyses establish which groupings belong to the same programming session (e.g. lab session) and grouping loggings for which we can be relatively sure there a no intermediate compiles which were not logged by the system.

The second part of this chapter focuses on introducing a new notion of generating views of the gathered data. Imagine wanting to perform an analysis which counts the number of errors per week, now imagine wanting to perform the same analysis, only students from each bachelor program individually. We have developed a method of presenting data for multiple figures for multiple groups of loggings while running the same analysis on each of these groups. This allows us to easily adopt a global analysis to specific group of loggings. An example would be to generate a figure which shows the overall amount of errors where each figure represents a group of loggings by the students grade.

4.2 Coherence between loggings

4.2.1 Session coherence

Van Keeken establishes the notion of coherence between log files on the basis of time spend between different compiles in section 4.2 of his thesis [HK07]. He established that declaring a generic time between compiles which can be seen as a single "session" is difficult, leaving it up to the developer to time limit which can be used to show coherence between files.

A number of factors influence whether two loggings should be considered to belong to the same session. First of all we expect a file to maintain the same filename. Next we expect loggings to result from the same student. As a final parameter we would like to take a look at how errors develop over time. But what is the correct parameter for the time between two loggings? The number of errors that are compiled within seconds from each other is quite large. But are students who do a next compile sixty minutes after the logging within the same session, what about 10 minutes?

We study the notion of a programming session by means of a number of analyses. Our first approach will be to look how long it takes for the number of new loggings to drop significantly. We approach this by simply grouping errors by the duration between them per student. We can define such an analysis like this:

```
coherenceTimeByUser = length
    <$$> (concat <$$> joinByKey <o> expandAnalysis <o> coherenceTimeByUser')
coherenceTimeByUser' = (mapAnalysis' coherenceTime) <o> perUserName

coherenceTime :: AnalysisKH [Logging] [Int]
coherenceTime = groupByAmmount <o> mapAnalysis' f <o> timeBetweenCompiles
  where f = ceiling <$$> divideByAna 60 <o> basicAnalysis_ timeDiffInSeconds

timeBetweenCompiles :: AnalysisKH [Logging] [TimeDiff]
timeBetweenCompiles = basicAnalysis "time between compiles" diff

divideByAna :: (Fractional a, DescriptiveKey key) ⇒ a → AnalysisFK key Int a
divideByAna n = basicAnalysis_ (λx → (realToFrac x) / n)

diff [ ] = [ ]
diff (x : xs) = tdiff' x xs
  where
  tdiff' x [ ] = [ ]
  tdiff' x (y : ys) = (logTimeDiff x y) : tdiff' y ys
```

In figure 4.1 we see that the majority of compiles happen within a few minutes within the previous compile. A safe assumption would be that any

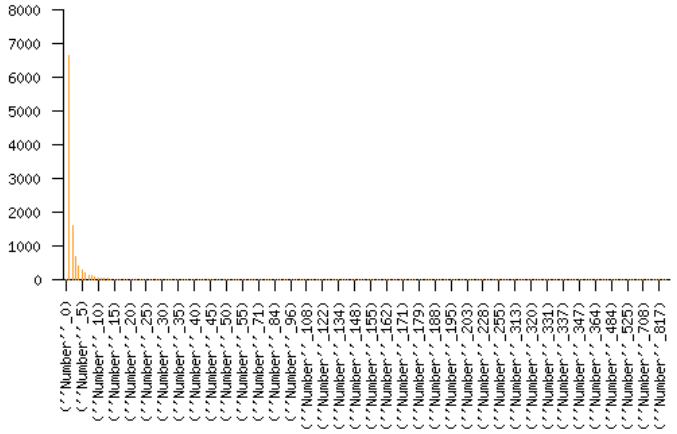

Figure 4.1: Frequency of time between compiles in minutes

compile below 10 minutes is within the same session, since there are hardly any two following compiles which have a larger time difference. This lead to our definition of *groupPerSession*.

```
groupPerSession :: AnalysisKH [Logging] [Logging]
groupPerSession = groupAnalysis_
  (λfstElem → ("Session", show $ logDate $ fstElem)) (groupBy isSession)
  where isSession a b = (toMinutes (logTimeDiff a b)) < 10
    toMinutes :: TimeDiff → Int
    toMinutes x = ceiling ((realToFrac $ timeDiffInSeconds x) / 60)
```

4.2.2 Completeness

For certain types of research completeness can be an important issue. A set of loggings is said to to be complete if no intermittent compilations were omitted. This can be caused be a number of different reasons. For example, a student may have took his work home, which is not logged by our system, after a number of compiles to continue the following day at the university. An example of an analysis in which completeness would be an issue is: Stating that users on average only add two line between compiles is only valid if we can be certain we have gathered all compiles. Otherwise they may have taken their work home, added 10 lines of code during development and recompiled their work at the university the following day.

We will therefore have to develop some form of grouping which bundles all compiled which seem to be a complete sequence. *How can this be determined?*

First of all we would need to take an approach which is quite similar to the one we took in section 4.2.1. We will need to make sure all the data we gathered belongs to a single and we'll need to make sure the data belongs is from the same file.

A new property would be the K-coherence property. We define the K-coherence property as follows: Logging a is k-coherent to logging b if no more than k lines of code in the sourcefile of logging a are different than those in logging b.

We will take a look at how K-coherence develops over the files. How many edits are there generally between two files? We will need to develop an analysis which is quite similar to the one provided for our studies into session coherence. We start by splitting up the loggings over the different students, next we split them up over different file names, to create the grouping on which we will perform our analysis.

```
fileAndStudent :: AnalysisKH [Logging] [[Logging]]
fileAndStudent = basicAnalysis "file and student" (groupBy f)
  where f a b = (isSameStudent a b) ∧ (isSameFilename a b)
```

Next we will need to retrieve the sourcefile and split the string over lines. We then compare the lines to the ones in the next logging, where next is defined by our grouping.

```
diffAnalysis :: AnalysisKH [[String]] [[(Data.Algorithm.Diff.DI, String)]]
diffAnalysis = mapAnalysis'
  (basicAnalysis_ (filter (λx → fst x ≢ B)))
   <o> basicAnalysis "diff" linesDiff

linesDiff [] = error "set to small for diff"
linesDiff (x : xs : []) = [(getDiff x xs)]
linesDiff (x : xs : xxs) = (getDiff x xs) : linesDiff (xs : xxs)
```

This leaves us with an analysis which returns the non matching lines, which we simply count.

```
diffSourceAnaLogs :: AnalysisKH [Logging] [Int]
diffSourceAnaLogs = mapAnalysis' setSizeAnalysis <o> diffAnalysis
   <o> selectGrpAnalysis "enough logs" (λx → length x > 1)
   <o> mapAnalysis' sourceLinesAna
```

This leaves us with lists of Integers, which indicate how many changes there were from compile to compile. Next we check need to count the frequency of each value. This indicates how often each number of line changes is encountered.

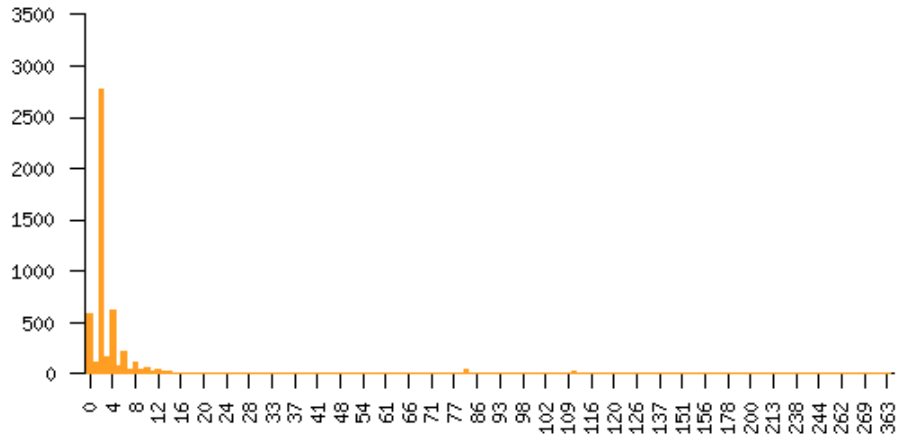

Figure 4.2: Frequence of number of line changes

```
freqNoDiffSourceLinesAna :: AnalysisKH [Logging] Int
freqNoDiffSourceLinesAna = setSizeAnalysis <○> groupPerValue <○>
  (concat <$$> mapAnalysis' diffSourceAnaLogs <○> fileAndStudent)
```

We can now view how many (line) edits are generally performed per logging in figure 4.2. This shows the majority of compiles show only 1 changed line. Of the 5349 differences 2769 showed only 1 changed line. If we choose to take a k value for our k-coherence of 10 we'll see that we end up with 4819 values, if we choose an k of 5 we end up accepting 4319 of the loggings as a complete set. This clearly shows the first few values carry the biggest weight.

The figure shows there are 588 compiles in which no edits were performed, due to the limitation of our setup we feared that these were cause by errors in imported source files. Fortunately while storing adding the compiler output to our log files we decided to include the exact source file with the messages reported by the compiler. To verify whether errors in included modules influenced our results we wrote an analysis. We checked the $sourceFile \circ logPath$ property against the file associated with each error.

To check whether the filenames in a logging are equal we wrote the following function.

```
isErrorSameAsFileName :: Logging → Bool
isErrorSameAsFileName log =
  ((sourceFile ∘ logPath) log) ≡ ((unPath ∘ fst ∘ head ∘ fnAndError) log)
unPath = last ∘ (map tail) ∘ (groupBy (λ_ x → x ≢ '/'))
fnAndError log = filter (λ(_, y) → length y > 0)
```

```
  $ map (g ∘ dropWarnings)
  $ f (compilerOut log)
  where f (_, files) = files
        g (a, bs) = (a, concatMap snd bs)
```

Where the *fnAndError* function returns all files which contained any errors, tupled with the errors encountered in that file. This should return one or zero values.

We used this function to create a new analysis, which counts the number of errors in which the file which contains the error does not match the file which was compiled.

```
isSameErrorAna :: AnalysisKH Logging Bool
isSameErrorAna = basicAnalysis "is same file" isErrorSameAsFileName

hasErrorInLog log = length (erFromLog log) > 0
erFromLog = (filter (λ(_, a, _) → a ≢ "Warning")) ∘ getErrorsFromOut ∘ compilerOut

numberOfNotSameFile :: AnalysisKH [Logging] Int
numberOfNotSameFile = numberOfFalse
  <$$> mapAnalysis'
    (isSameErrorAna <○> selectGrpAnalysis "hasErrors" hasErrorInLog)

numberOfFalse :: [Bool] → Int
numberOfFalse = foldl (λa b → a + fromBool b) 0
  where fromBool True = 0
        fromBool False = 1
```

Fortunately performing this analysis showed us there were 0 loggings which contained errors in the imported modules.

We use the previous research to introduce the *groupPerKCoherence* grouping. We feel that 5 would be an acceptable k value. This indicates a small change in the overall source file. For example, 5 lines could easily be interpreted as writing one or two functions.

```
groupPerKCoherence :: Int → AnalysisKH [Logging] [Logging]
groupPerKCoherence k = groupAnalysis filename
  (groupBy (isKCoherent k))

filename = sourceFile ∘ logPath
isKCoherent k a b = (isSameFilename a b)
  ∧ (isSameStudent a b)
  ∧ ((lineCoherence a b) < k)
```

4.3 Presenting Neon results

Van Keeken established a format for manipulating data into a number of figures and tables. Analyses using multiple figures where presented in either a TEXfile or an Html document. Html combinators in Haskell allowed Van Keeken to add text to documents, for writing complete reports. He developed combinators for TEXwhich allowed him to present data in pdf format.

Unfortunately this approach does not scale very well. Writing large documents within Haskell, even with combinators, is an arduous process, requiring recompilation and the re-running of the required analyses on a large dataset. Each (minor) edit means repeating this process.

We have therefore chosen to adopt a more ghci driven development cycle.

4.3.1 Presenting data in multiple figures

Presenting a large amount of figures based on the same data originally required the writing of a large number of individual analyses, in which certain data was filtered in each run. They would be combined using the (+++) combinator. We developed an easier method to differentiate over different types of analyses.

As noted in chapter 3 the *Analysis* type (ignoring context) is $[(keya, a)] \rightarrow [(keyb, b)]$. The initial type of the key-value pair after parsing is $[(KeyHistory, [Logging])]$. Initially containing a single value, with a single key with all parsed loggings as the value. We cannot get any guarantees about the eventual result type of our analysis, so splitting up the analysis over different values will have to be done before performing any analysis.

To achieve this type of splitting of analyses we introduce a new operator. The <++> combinator is defined as follows:

```
(<++>) f a = multiFigureAnalysis f a
multiFigureAnalysis :: (Show c, DescriptiveKey key, Ord c) ⇒
    (a → c) → AnalysisFK key [a] b →
    [(key, [a])] → [(String, [(key, b)])]
multiFigureAnalysis f ana = (λ((key, logs) : _) →
    (map (λx → ((fst x), (ana <$> (snd x)))) (distValues' f key logs)))
  where
    distValues' f key logs = map
      (λx → (((show ∘ f ∘ head ∘ snd ∘ head) x, x)))
      (distValues (comparing f) key logs)
```

```
distValues f key logs = listPerEntry (zip (repeat key) (apOrd f logs))
  where apOrd f = (groupBy (λx y → (f x y) ≡ EQ)) ∘ (sortBy f)
    listPerEntry = map (λx → [x])
```

Applying this to our regular analysis type, which is *AnalysisKH* [*Logging*] *a* this will give us a combinator of the type:

```
(<++>) :: (Logging → c) → AnalyisKH [Logging] a →
  [(KeyHistory, [Logging])] → [[(KeyHistory, b)]]
```

The type of this combinator does not allow us to add any new combinators once it has been applied. The combinator needs to be applied afterwards. It can be applied to any available analysis however, since it will alter the input of the analysis, not the output.

A simple example of a multi analysis creating function using this combinator would be:

```
figurePerWeek :: AnalysisKH [Logging] b →
  [(KeyHistory, [Logging])] → [(String, [(KeyHistory, b)])]
figurePerWeek = (<++>) (weekOfYear ∘ logDate)
```

Which will allow us to perform any analysis, presented in a per week manner.

We could apply the *figurePerWeek* analysis to the following:

```
setSizeAnalysis <∘> groupPerStudent
```

This will yield a list of result sets, which unfortunately cannot be presented using the regular functions, since they only work on a single result set. To present the results of these analyses we define simple mapping function.

```
renderFunctions :: FilePath → (a → (FilePath, Plot)) →
  [(String, a)] → IO [FilePath]
renderFunctions p f = mapM (renderFunction' p f)
  where renderFunction' p f (x, y) = do
    file ← renderPNGPlot p (fst $ f y) (snd $ f y)
    let newname = joinToFp $
      addToLast ((stringToFilePath x) ++ "_") $ unFilepath file
    system ("mv " ++ file ++ " " ++ newname)
    return newname
unFilepath xs = map tail $ groupBy (λ_ x → x ≢ '/') xs
addToLast _ [] = error "addToLast: empty list"
addToLast s (x : []) = return (s ++ x)
addToLast s (x : xs) = x : addToLast s xs
```

joinToFp = *concatMap* ((:) '/')

Applying *renderFunctions* to our result sets now yields a list of results, an excerpt including four weeks can be seen in figures 4.3, 4.4, 4.5 and 4.6.

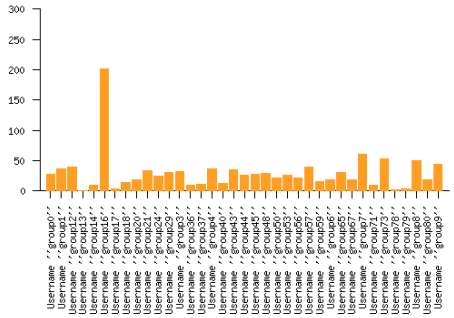

Figure 4.3: Number of Loggings per student in week 6

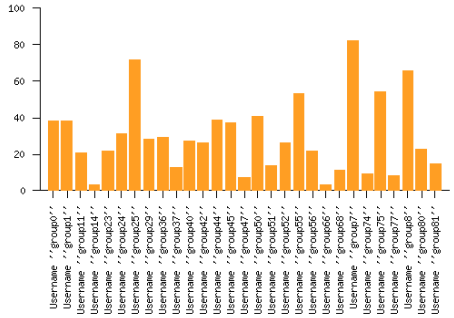

Figure 4.4: Number of Loggings per student in week 7

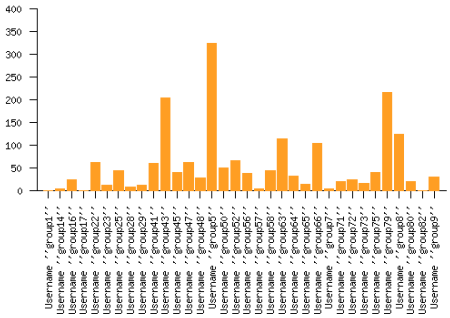

Figure 4.5: Number of Loggings per student in week 8

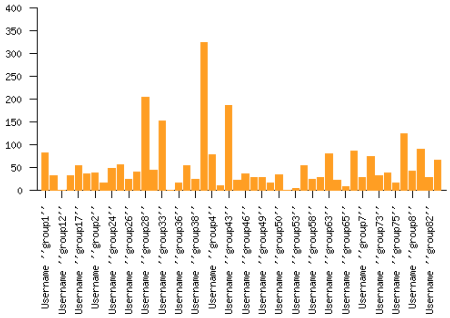

Figure 4.6: Number of Loggings per student in week 9

Chapter 5

Investigating Frequent Flyers

5.1 Introduction

This chapter focuses on developing a view of which error messages are presented to the user. Helium can generate a number of different kinds of errors. For example, parse, lexical and type errors. These error kinds generate different error messages, which are presented to the user.

In this chapter we present two kinds of analyses. The first globally establishes which errors are most frequently encountered. The second is based on the frequently encountered error in the first and studies how these frequently encountered errors develop over time.

Before investigating which type of errors are interesting to study it is best to get some sort of overview of the type of errors we have to choose from. It would be unwise to study the effects of errors rarely encountered for a number of reasons. First of all, they are unlikely to allow us an unclouded view of the development of errors over time, since our results will be biased by having a very small set of data from which we collect data. To develop a sound view of how errors develop we need a sufficiently large amount of data, loggings, meta-data and individual students to develop our views from. For example, if we choose to investigate an error which is only rarely encountered and we find at a later point in time all our results were based on a very select group of students it is very unlikely that our results are valid for anyone but this select group of students. Secondly rarely encountered errors are not that interesting.

5.2 Approach

To develop a view of the kind of errors students encounter we needed to extend Neon as described in section 3.2.3. Neon allows us to take a look at the loggings provided by Helium. These loggings contain the details to see when Helium ended compilation and how (e.g. by failing) but does not allow us to distinguish which errors where encountered.

The loggings provided the exact version of the compiler with which the submitted sources were compiled. These versions will no longer work in binary form on modern Linux / Unix machines as used to develop Neon. The source version is also no longer compilable on the modern Haskell platform. Fortunately, slightly modified variants of the original sources have been maintained which can be compiled with (currently) Ghc 6.10. We decided to use the recompiled source files provided by Van Keeken by using these maintained versions of Helium.

We were able to use these recompiled versions to collect each individual error message provided to a student, their exact location, the compilation phase from which they originated and the exact text presented to the student. Unfortunately Neon was yet unable to handle these types of data. We therefore extended Neon to include a large number of basic analyses on this data. Analysing Neon's error output gave us two for the price of one, as it also allowed us to collect data about the warnings presented to students.

5.3 Results

We looked at in which frequency which error is presented to students. This allows us to gain an insight into which kinds of errors are encountered frequently by students. An excerpt of the results of this analysis is shown in figures 5.1 and 5.2. They show the top 10 error messages which were presented to students, and the number of times they were presented. The top 10 covers 2446 of the total amount of 8607 errors. These figures were generated by the *topNOMessagesPerWeek* analysis below.

```
nOMessagesPerWeek = basicAnalysis "number of" length
    <o> expand'
    <o> (mapAnalysis getErrors)
    <o> groupPerWeek
topNOMessagesPerWeek = topN 10
    <o> nOMessagesPerWeek
```

Note that the definition of *expand'*, *getErrors* and *topN* can be found in appendix A.3.

Syntax error	394
Type error in literal	102
Type error in variable	211
Bracket '(' is never closed	89
Type error in application	673
Type error in right-hand side	242
Type error in infix application	451
Type error in pattern application	111
Close bracket ')' but no open bracket	78
Type error in explicitly typed binding	95

Figure 5.1: Top 10 encountered errors by message

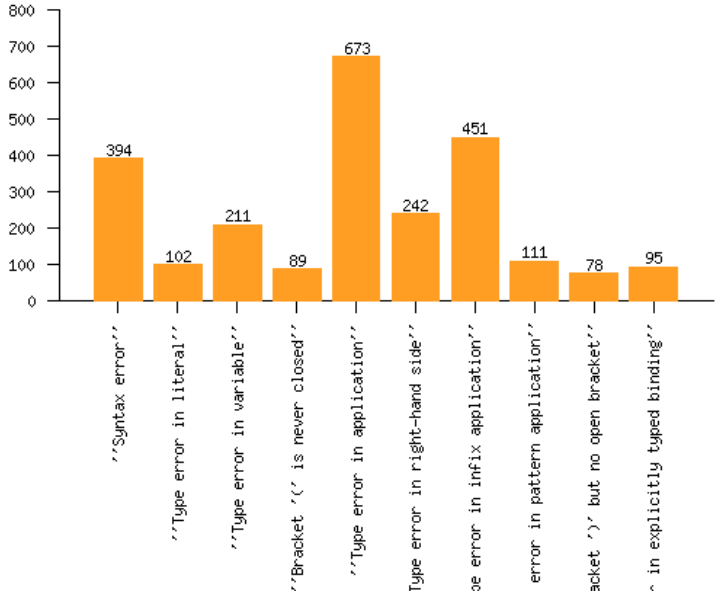

Figure 5.2: Top 10 encountered errors by message (Graphically)

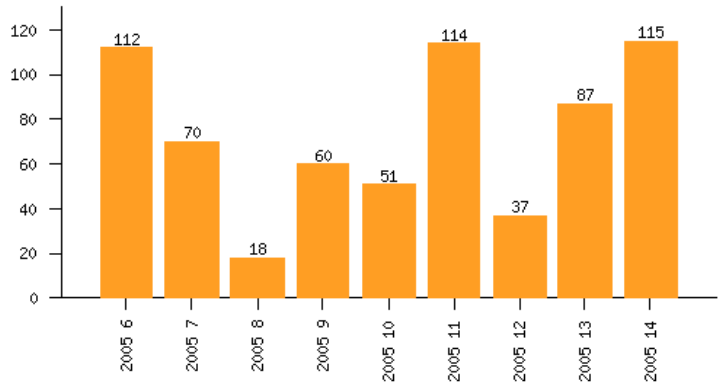

Figure 5.3: Type error in application per week

This defines a basic view of which errors seem most interesting to us. We would like to develop a clear view of how these errors develop over time. *Does the functional programming course influence their frequency? Do students develop a better understanding of the language, enabling them to make less errors of this specific type?* To answer these questions we have distributed the error messages over time.

We take a look at the distribution of type errors per week, as they seem to be encountered most frequently. The most commonly produced message of this type is the message "Type error in application". We take a look at how it develops over time in figure 5.3 This figure is produced by the following analysis:

```
typeErrors = (filter (λ(_, a, _) → a ≡ "Type error in application"))
    <$$> getErrors
nOTypeErrorsPerWeek = basicAnalysis "number of type" length
    <o> expand'
    <o> (mapAnalysis typeErrors)
    <o> groupPerWeek
```

These figures actually don't tell us an awful lot. It is quite possible that the amount of errors is directly related to the total amount of collected loggings, as shown in figure 5.4. We can however easily compare these results to the amount of loggings submitted per week, as presented in figure 5.5

```
averageErrorAnaByName x = averageAnalysis
    <o> mapAnalysis (x <?> length <$$> f)
    <o> groupPerWeek
```

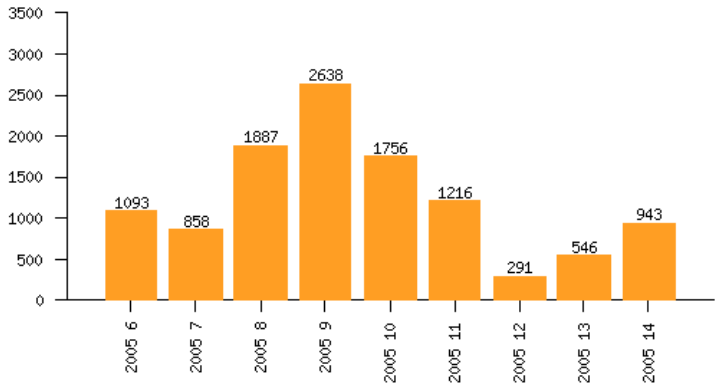

Figure 5.4: Total amount of loggings per week

```
    where f = (filter (λ(_, a, _) → a ≡ x)) <$$> getErrors
numberOfTypeErrorInApplicationsPerWeekPerCompile =
    averageErrorAnaByName "Type error in application"
```

We can easily do the same for the other error types, like "Type error in infix application" as seen in figure 5.7 or "Syntax error" as seen in figure 5.6. These too allow us to study the development of these error types over time.

```
averageTypeInInfixPerWeek =
    averageErrorAnaByName "Type error in infix application"
averageSyntaxErrorPerWeek =
    averageErrorAnaByName "Syntax error"
```

Overall some figures actually show an increase in the relative amount of errors over time, while the overall amount of errors is actually erratic as seen in figure 5.8. It is highly unlikely that this means students are actually getting worse at writing Haskell programs. What's more likely is that there are external factors influencing their grades. It is in the following chapters that we take a look at which factors influence these erratic behaviour in the amount of submitted errors and which factors influence their distribution over time.

The complete data, as provided in the datasets, shows a large amount of errors which are only encountered once. These are generally messages indicating some function has been applied to the wrong type of arguments. These are generally either caused by introducing undefined variables or applying undefined constructors. The actual name of the variable is supplied in the error message, causing them not to be grouped together. If we sum

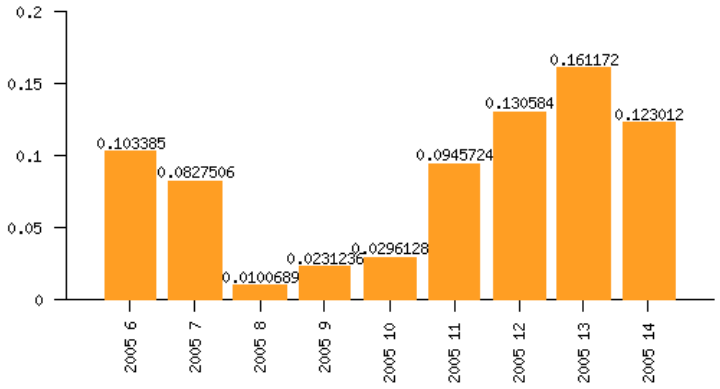

Figure 5.5: Type error in application per week

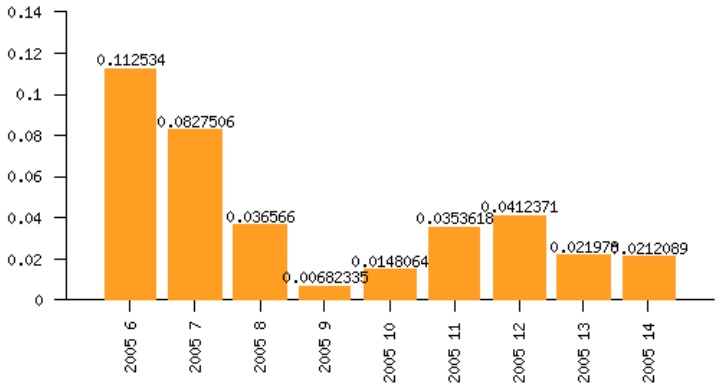

Figure 5.6: Syntax error per logging per week

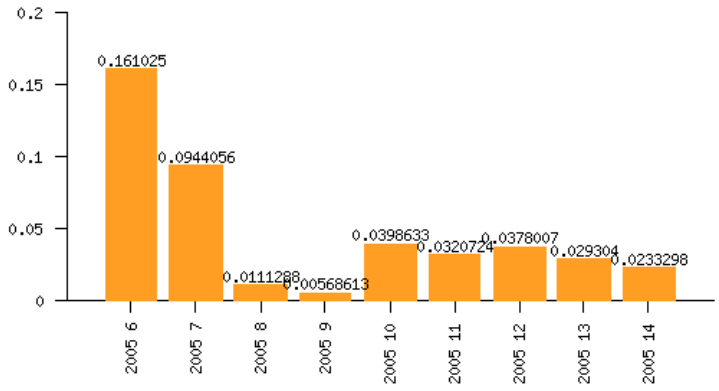

Figure 5.7: Type error in infix application per logging per week

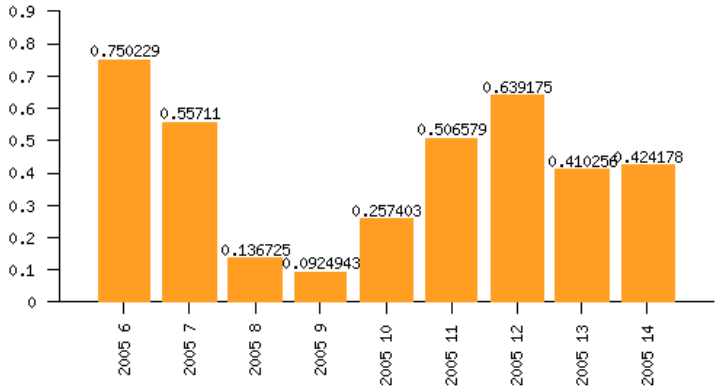

Figure 5.8: Errors per logging per week

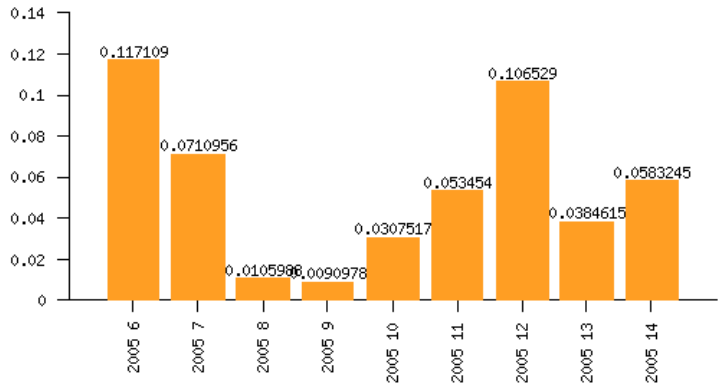

Figure 5.9: Undefined variables per logging per week

this data up we actually get some numbers which rival those presented in our top 10. We took a look at the number of undefined variables in figure 5.9. This shows, for example a higher number of "Undefined variables" than "Syntax errors".

```
averageUndefinedErrorsPerWeek = averageAnalysis
    <o> mapAnalysis (str <?> length <$$> f)
    <o> groupPerWeek
  where f = (filter (λ(_, a, _) → (intersect str a) ≡ str))
      <$$> getErrors
    str = "Undefined variable"
```

Chapter 6

Investigating Frequent Warnings

6.1 Introduction

In this chapter we investigate how warnings develop over time. Warnings do not need to be fixed before being able to run a program, so they are allowed to survive many compiles. Their patterns therefore differ from those seen in errors.

Good warnings and hints are seen as one of the key features in Helium [HLvI03]. Unfortunately there have never been any studies on the effects of these "good" warnings. It therefore remains unclear how students respond to these warnings. Our extensions to Neon allow us to study the effect of Helium's warnings and hints by monitoring the response of a student to a particular warning message.

There is a wide range of warning messages Helium can throw. For example, the function shown in 6.1 makes the compiler report 6.2 and the function in 6.3 makes the compiler report 6.4.

We will perform two kinds of analyses in this chapter. Our first analyses shows how long warnings live. Our second analysis investigates how warnings are threated around deadlines.

$$fac\ n = product\ [1\,..\,n]$$

Figure 6.1: code example

```
(1,1): Warning: Missing type signature: fac :: Int -> Int
```

Figure 6.2: Helium result

$$
\begin{aligned}
&\mathit{trivial} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{trivial}\ a \mid a < 10 = 1 \\
&\qquad\qquad \mid a \geqslant 10 = 2
\end{aligned}
$$

Figure 6.3: Code example (2)

6.2 Approach

Taking a look at warnings is somewhat different from looking at errors. Students are allowed to keep developing their programs after having been presented a warning. They won't be stopped from developing a more complex problem until the warning is solved. There is also the added difficulty of some warnings being created by intentional actions by the programming. We have to envision some kind of method to isolate a lifespan of each individual warning message as it travels through different phases of compiles.

Achieving this required us to introduce a new method of grouping loggings. This grouping would have to be able to monitor a message over different logging entries. An added difficulty is the fact that the amount and type of warnings per logging may vary greatly.

The new type of grouping therefore replicates the original logging for each warning it encounters in a certain compile and returns the warning combined with the original logging. This allows us to take a look at the results of an analysis for each individual warning presented to a student, giving us the possibility to monitor the consequences of any given message.

6.2.1 Duration of warnings

We looked at how long warnings last. *Do students act on hints and warnings like they are errors? Instantly solving them like would be obligatory with errors.*

```
(2,11): Warning: It is good practise to have 'otherwise' as the last guard
```

Figure 6.4: Helium result (2)

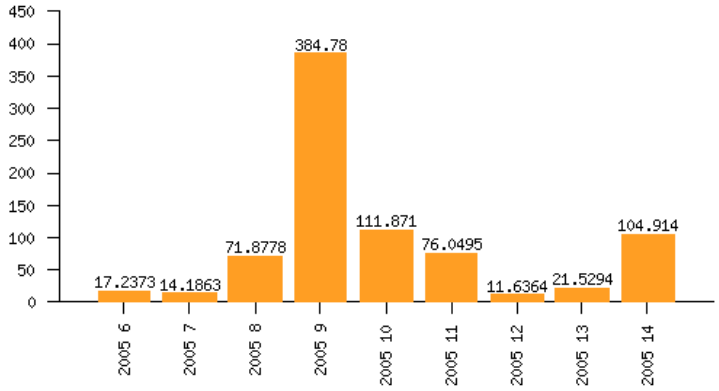

Figure 6.5: Average duration of warnings per week in number of minutes

We doubt they do, but we'll take a look at the numbers.

durationOfWarnings = `"duration of warnings"`
<?> *mapAnalysis* (*basicAnalysis_fst*)
<o> *groupPerWeekTupled* <o> *aggregateAnalysis* `"joined"` *concat*
<o> *joinByKey* <o> *distance* <o> *ungroupedEqWarnings*

averageDurationOfWarnings = *averageAnalysis*
<o> *durationOfWarnings*

avgDurationOfWarningsInMinutesPerWeek = *averageAnalysis*
<o> *durationOfWarningsInMinutesPerWeek*

durationOfWarningsInMinutesPerWeek = *durationOfWarningsInMinutes*
<o> *groupPerWeek*

durationOfWarningsInMinutes = `"duration of warnings in minutes"`
<?> *mapAnalysis* (*basicAnalysis_fst*) <o> *groupPerWeekTupled*
<o> *aggregateAnalysis* `"joined"` *concat*
<o> *joinByKey* <o> *distanceInMinutes* <o> *ungroupedEqWarnings*

As we see in figure 6.6 it sometimes takes quite long for students to fix a warning. *Why does the time it takes students to fix take these warnings this long?* We do not know.

6.2.2 When do warnings end

Since the duration of errors seems erratic we will have to take a look at some other properties. Students are required to hand in exercises at a certain deadline. We expect to see a sharp rise in the amount of warnings

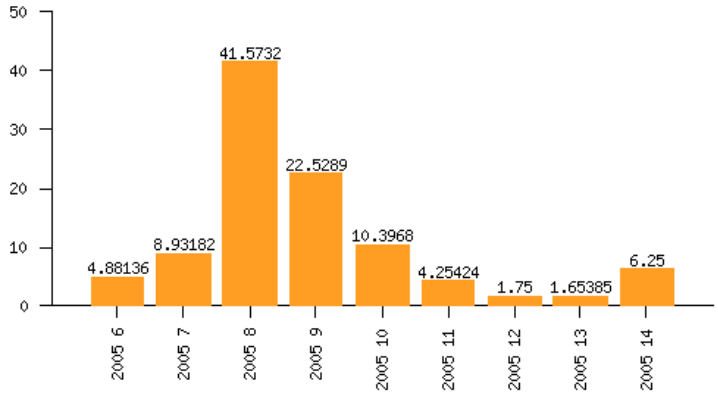

Figure 6.6: Average duration of warnings per week in number of compiles

fixed near a deadline. Writing code which generates a lot of warnings at compile time is generally recognized as a bad coding practice. Warnings therefore may be believed to have a negative influence on a students grade. We therefore expect students to rapidly fix these errors before submitting their work.

We can largely develop such an analysis from the ones described in the previous section, but we'll need to write a few custom lines of code. First of all we'll need to able to filter out which loggings where submitted in a certain week. We achieve this with the following analyses.

```
getWeekN :: Year_Week → AnalysisKH Logging Logging
getWeekN n = selectGrpAnalysis
  ("week " ++ show n) (λlog → ((weekOfYear ∘ logDate) log) ≡ n)
```

The Year_Week datatype was provided by Van Keeken and is generally used for grouping per week. We can ofcourse easily use it to filter too, using the same functions we normally would to build such a grouping.

Next we will need to group the loggings per day:

```
monthDay :: CalendarTime → (Month, Int)
monthDay x = (ctMonth x, ctDay x)
dateOfWarnings a b = monthDay ∘ fst ∘ fst <$$> lastAna
  <○> ungroupedEqWarnings
  <○> mapAnalysis' (getWeekN (Year_Week a b))
groupPerMonthDay :: AnalysisKH (Month, Int) [(Month, Int)]
groupPerMonthDay = superGroupByOrd (λ(x : xs) → show x) compare
```

It is important that we group the errors by the latest timestamp in the

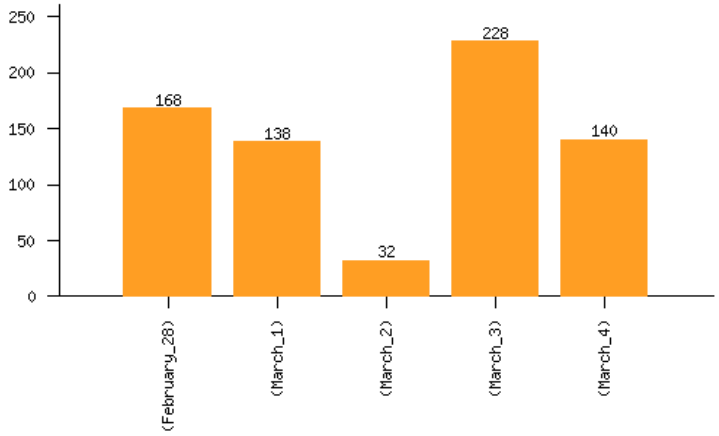

Figure 6.7: Number of warnings ending each day near a deadline (4-3-2005)

set, since otherwise we cannot be certain about the end date of a certain warning.

Lastly we need to combine the analyses and calculate the length of each set to see how many warnings ended each day.

$$noEndingsPerDate\ y\ w = length \\ <\$\$>\ groupPerMonthDay\ <\circ>\ dateOfWarnings\ y\ w$$

We performed this analysis for $y = 2005$ and $w = 9$, week 9 of 2005, the deadline of the first assignment for students was march 4th. The results of this analysis can be seen in figure 6.7. Next we did the same for week 14, on which a deadline was present on the 9th of april. The results of this analysis can be seen in 6.8.

It is interesting to note the sharp decrease in number of fixes warnings at both march 2nd and april 6th. This is most likely caused by the fact that the course classes (two times two hours) and practical sessions (four times two hours, lab and exercise sessions) were planned on mondays, tuesdays, thursdays and fridays.

We expected the number of absolute warnings fixes to increase near a deadline, but apparently they do not. We expect that at least the relative number of warning fixes will increase over time. We have developed the following analysis to investigate this.

$$noEndingsPerDayRel\ y\ w = divideAna \\ <\circ>\ ((noEndingsPerDate\ y\ w)\&>\ (noLoggingsPerDay'\ y\ w)) \\ noLoggingsPerDay'\ y\ w = length\ <\$\$>\ groupPerMonthDay\ <\circ>\ f$$

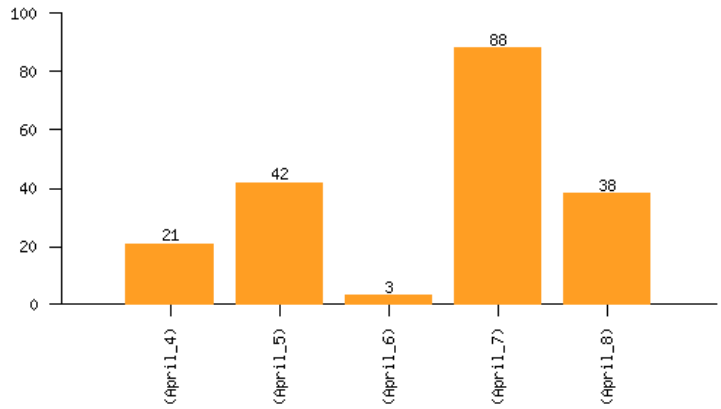

Figure 6.8: Number of warnings ending each day near a deadline (9-4-2005)

where $f = (monthDay \circ logDate)$
<$$> *expandAnalysis*
<○> $mapAnalysis'$ ($getWeekN$ ($Year\_Week$ y w))

The results of this analysis for weeks 9 and 14 can be seen in figures 6.9 and 6.10.

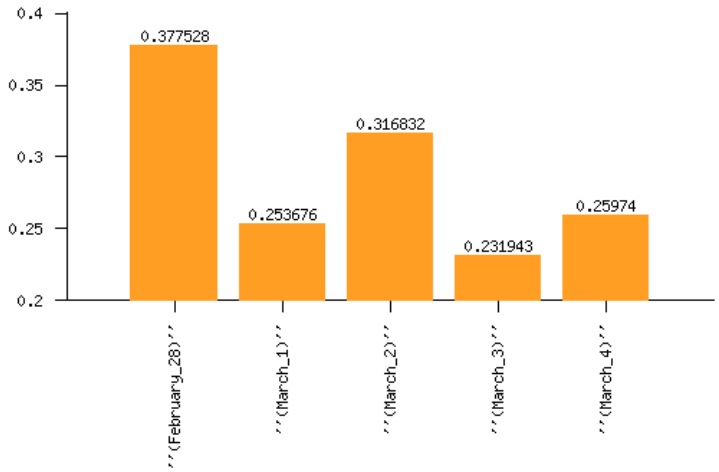

Figure 6.9: Relative number of warnings ending each day near a deadline (4-3-2005)

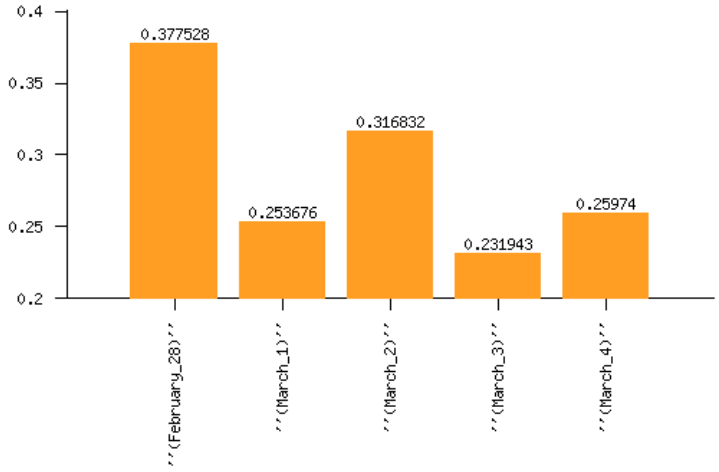

Figure 6.10: Relative number of warnings ending each day near a deadline (4-3-2005)

Chapter 7

Parse Errors

7.1 Introduction

This chapter focuses on how parse errors develop over time and patterns in their development. The Helium compilation process contains a number of distinct phases which can throw errors. When compiling a Haskell source file Helium goes through a number of distinct and clearly separate phases, with their own purpose. These respectively are the lexical, parse, resolve operator, static, typing and code generation phases. The phase we will be looking at in this chapter is the parsing phase.

Initially a file is divided into tokens by the lexical phase. The goal of the parse phase is to combine these tokens into a legal parse tree. A parse error therefore indicates that all symbols inside a file are valid symbols within the Helium system, but their sequence is not correct or incomplete. An example of a possible parse error can be seen in figure 7.1. This chapter build on the research into "Syntax errors" in chapter 5, as the compiler reporting a "Syntax error" indicates a parse error occurred.

In this chapter we have done two types of analyses. The first focuses on amount of parse errors students make over time. The second focuses on which kind of parse errors students encounter.

7.2 Claim

As in previous chapters we expect students to become more proficient at programming Haskell as the course progresses. Initially students struggle with valid ranges of tokens which make up a language, they are not aware

```
Compiling ./Test.hs
(4,9): Syntax error:
    unexpected '='
    expecting operator, '->' or '|'
Compilation failed with 1 error
```

Figure 7.1: example of a parse errors

of all available constructs and confuse language constructs based on their experience in other languages. Over time they develop a more thorough understanding of the language, which includes a knowledge of the proper syntax. We therefore expect to see fewer parse errors over time.

7.3 Approach

Our extensions to Neon allow us to explore the exact compiler output presented by Neon. This allows us to view the messages thrown by the compiler during the parsing phase, indicated as a syntax error by Helium. A parse error always contains an exact location at which the parser failed at consuming the tokens which were provided by the lexical phase.

Unfortunately our approach as used in chapter 5 has some limitations. The message printed by the compiler does not indicate the exact structure on which the compiler failed. It just mentions which token it encountered and which is expected. For example, a student writing a type will have used an $\rightarrow$ in the type signature, but this same token is also used in case statements. A function like:

$$
\begin{aligned}
&a :: \mathit{String} \rightarrow \mathit{String} \\
&a\ x = \textbf{case}\ x\ \textbf{of} \\
&\quad \_ \rightarrow \texttt{"a"} \\
&\quad \texttt{"a"} = \texttt{"b"}
\end{aligned}
$$

will throw an error when encountering the = operator and Helium will report a message as seen in figure 7.1. This does not indicate something went wrong inside the case expression, rather giving a generic message explaining that the = token is invalid at this point.

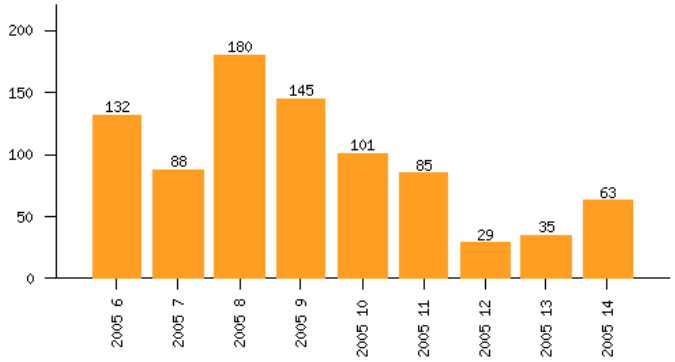

Figure 7.2: Number of parse errors per week

7.4 Research

7.4.1 Amount of parse errors

We will start by taking a look at the amount of parse errors in logged source files per week. Studying the amount of parse errors can be achieved by retrieving all errors from the parse phase, as stored in the *CompilerOut* structure of the logging. All we then need to do is calculate the size of the set. We have grouped this analysis per week.

```
numberOfParseErrorsPerWeek = "number of parse errors"
    <?> length <$$> parseErrorsPerWeek
parseErrorsPerWeek = concatAnalysis
    <o> mapAnalysis parseErrors
    <o> groupPerWeek
parseErrors = (getErrorsBy (λ(phase, _) → phase ≡ Parsing))
getErrorsBy :: (CompiledPhase → Bool) →
   AnalysisKH Logging [Error]
getErrorsBy f = filterWarnings
    <o> basicAnalysis "Errors" (getErrorsFilterPhase f)
    <o> getCompilerOutput
```

The results can be seen in 7.2, where we see the number of parse errors per week divided by the total amount of errors per week.

The relative amount of parse errors can be determined by dividing the amount of parse errors by the total amount of errors per week.

```
partParseErrorsPerWeek =
   (λ(x, y) → (realToFrac x) / (realToFrac y))
```

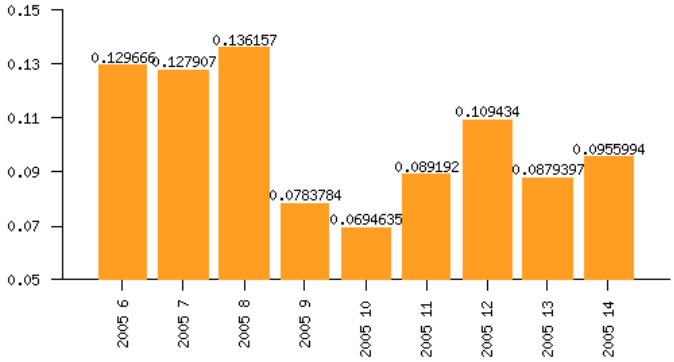

Figure 7.3: Relative number of parse errors per week

```
    <$$> parseErrorsAndTotalErrorsPerWeek

parseErrorsAndTotalErrorsPerWeek = (f parseErrors)
    <&> (f getErrors) <○> groupPerWeek
  where f ana = length <$$> concatAnalysis
    <○> mapAnalysis ana
```

the results of this analysis can be seen in figure 7.3, where we see the number of parse errors per week divided by the total amount of errors per week.

These figures do seem to indicate that the amount of parse errors encountered by students goes down significantly after the first three weeks of the course. After there three weeks the amount seems to level out.

Taking a look at this analysis over different types of students results in an analysis which looks like the following:

```
logStudentType :: Logging → Maybe StudentType
logStudentType x | studentData x ≡ [ ] = Nothing
    | otherwise = (studentType ∘ head ∘ studentData) x

partParseErrorsPerStudentType = logStudentType <++> partParseErrorsPerWeek
```

This leads to the results which are seen in figures 7.4, 7.5, 7.6 and 7.7. These result deserve some explanation. First of all the number of NAST (Physics) students was only 1. He did finish the course, but apparently did most of his work at home. There is very limited amount of loggings done by CKI (cognitive artificial intelligence) students in weeks, since most of them stopped the course on may 22nd, some however, did follow the full computer science course. This limited amount of data may cloud the results in later weeks, from week 11. The amount of math (WISK) students was also somewhat limited, only five math students enrolled in the course.

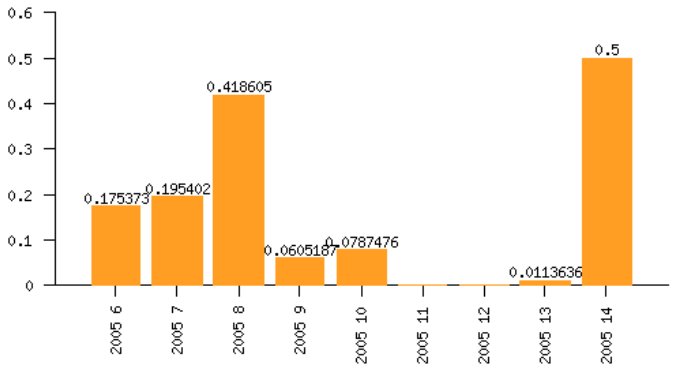

Figure 7.4: Relative number of parse errors per week (CKIB)

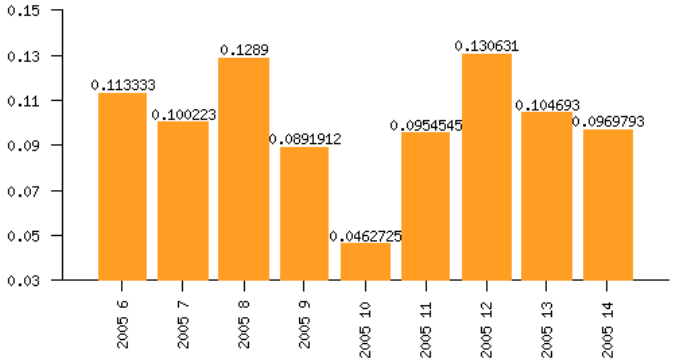

Figure 7.5: Relative number of parse errors per week (INCA)

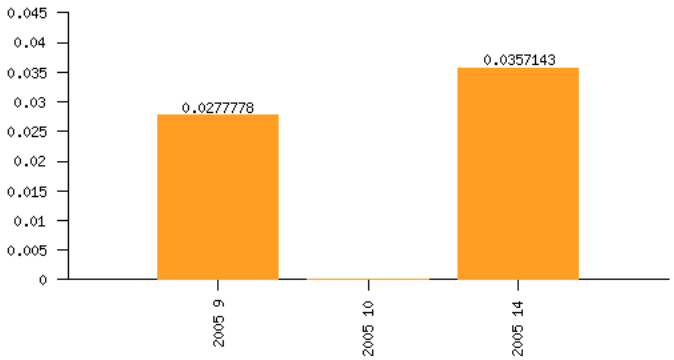

Figure 7.6: Relative number of parse errors per week (NAST)

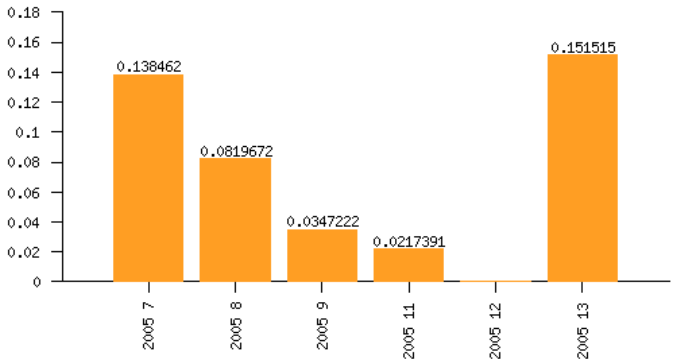

Figure 7.7: Relative number of parse errors per week (WISK)

7.4.2 Kinds of parse errors

It might be interesting to see in which kind of constructions students make most errors. For example, *did the student attempt to write a type or a function?*

In section 5.3 we viewed the amount of parse errors per week we would now like to view that data in a more detailed manner, but described in section 7.3 we cannot easily differentiate the kind of parse errors students make. The parser does not indicate which kind of construction it attempted to construct. We therefore had to conceive some way of harvesting this information.

Initially we started with selecting the files which contain a parse error. This can be done by combining the data from the parse errors and the data from the initial logging. The initial *Logging* contains a *path* field which stores the path. We combined this data with the parse errors in the data, in the following analysis:

$$
\begin{aligned}
&\mathit{parseErrorsPerTypeFile}' = \mathit{map}\\
&\quad (\lambda(x,y) \rightarrow (\mathit{filename}\ y, \mathit{location}\ x))\ \texttt{<\$\$>}\ \mathit{parseErrorsPerType}'\\
&\quad \textbf{where}\ \mathit{location}\ (a,\_,\_) = a\\
&\qquad \mathit{filename}\ \mathit{log} = \mathit{toFilePath}\ (\mathit{logPath}\ \mathit{log})\\
&\mathit{parseErrorsPerType}' = \mathit{selectGrpAnalysis}\ \text{""}\ (\lambda x \rightarrow \mathit{length}\ x > 0)\\
&\qquad \texttt{<o>}\ \mathit{concatAnalysis}\ \texttt{<o>}\ \mathit{repliZip}\\
&\qquad \texttt{<o>}\ \mathit{mapAnalysis}'\ (\mathit{parseErrors}\ \texttt{<\&>}\ \mathit{id})\\
&\mathit{parseErrorFilesPerStudent} = \mathit{parseErrorsPerTypeFile}'\ \texttt{<o>}\ \mathit{groupPerStudent}
\end{aligned}
$$

The result of this analysis is a list of file names and positions at which the error was thrown, as reported by the Helium compiler. We stored this data in a spreadsheet, a tab separated file, as can be seen in figure 7.8.

```
/home/mathijs/datasets/fp-0405-p3//group3/2005-02-07@16_02_22_935/Hello2.hs        ("2","25")        keyword as function
/home/mathijs/datasets/fp-0405-p3//group3/2005-02-07@16_02_39_517/Hello2.hs        ("2","25")        keyword as function
/home/mathijs/datasets/fp-0405-p3//group3/2005-02-07@16_03_00_429/Hello2.hs        ("2","25")        keyword as function
```

Figure 7.8: excerpt from tsv file

We required a way to extract more information from the stored source files. Especially about the kind of construct in which students made a particular error. A possible approach is to adopt the Helium parser to output more concise information about the actual position in which it failed. This approach failed however, since the parser never reports the exact language construction on which it failed. This is because of the nature of Parsec's parser combinators, which always report the actual parser inner most parser that failed. This means that the Helium parser will fail on the tokens it is trying to match, not on the constructor in which scope it is trying to match the next token. It would of course be possible to adopt the parser to function in such a way, but that would not be trivial. We therefore chose another approach to solving this problem.

Eventually we took a different approach to categorising the errors presented to students. Heliums logging facility has gathered all original source files, allowing us to view the source code. The total amount of parse errors, made over the 2004-2005 course, sums up to a little under 900 errors. This is just within the amount of source files which can be analysed by hand, without resorting to an unrealistic amount of manual labor. We have therefore indexed the error messages in each source file by hand. The upside of this approach is the fact that we have been able to do so quite accurately.

Note that we decided to group the data per student, as to be able to browse the loggings per student. This has the benefit of being easily able to identify recurring mistakes by students, allowing us to traverse the source files more rapidly.

We stored the data we have gathered manually using this approach in the *courseData* section of the original *Logging*. We extend the *courseData* field of a loging with the *parseErrorData* property. This is where we store the kind of parse error a student makes.

The analysis we have created on the basis is the *pErrorKindAna* which counts the number of errors per parse error kind.

```
pErrorKindAna :: AnalysisKH [Logging] Int
pErrorKindAna = setSizeAnalysis
    <o> groupPerParseErrorKind
    <o> mapAnalysis' (sel <o> selectJust)
    <o> relParseErrorType'
```

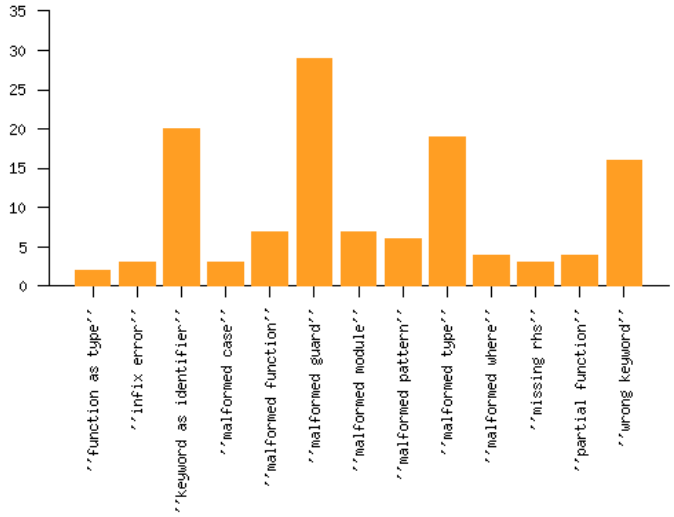

Figure 7.9: Number of parse errors per kind (Week 6)

```
groupPerParseErrorKind :: AnalysisKH [String] [String]
groupPerParseErrorKind = groupAnalysis id (group ∘ sort)
```

The *groupPerParseErrorKind* groups by parse error kind.

```
sel = selectGrpAnalysis "" ((≠) "") <○> selectGrpAnalysis "" ((≠) "??")
selectJust :: AnalysisKH (Maybe a) a
selectJust = fromJust <$$> selectGrpAnalysis "Just" isJust
```

The *selectJust* works on maybe data and returns the values from the justs. The *sel* analysis removes all values which we were unable to classify.

We use the <++> operator to generate a separate for each week.

```
pErrorKindAnaPerWeek = (weekOfYear ∘ logDate) <++> pErrorKindAna
```

The result of this analysis can be seen in figures 7.9, 7.10, 7.11, 7.12, 7.13, 7.14, 7.15, 7.16 and 7.17. As we see, one of the most encountered issues is that the right hand side of a function is missing (indicating that function was correct up to the = token). This may indicate a student tested if a file compiled up to the point of the error. We also note that students make many errors using guards, by for example using the = before the bar, at first, but by week 8 this issue is rarely encountered.

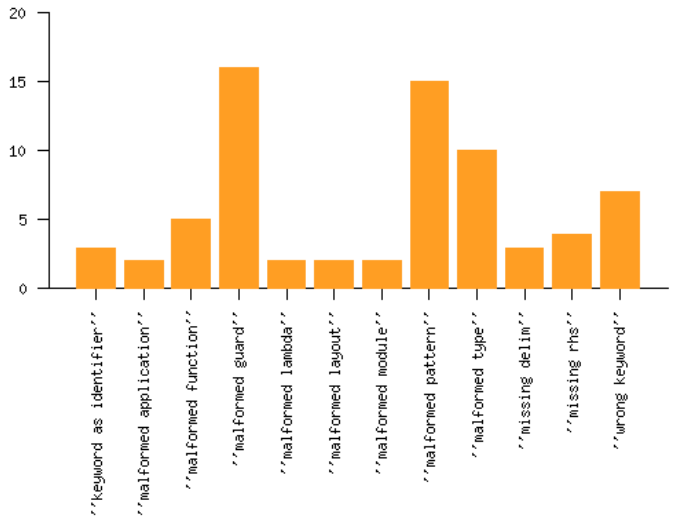

Figure 7.10: Number of parse errors per kind (Week 7)

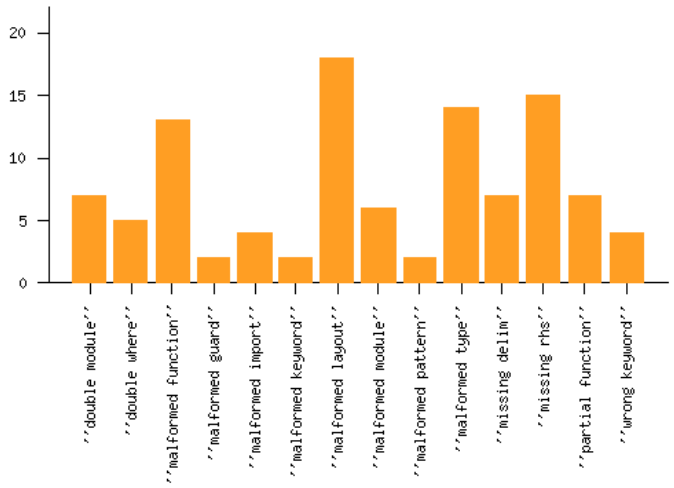

Figure 7.11: Number of parse errors per kind (Week 8)

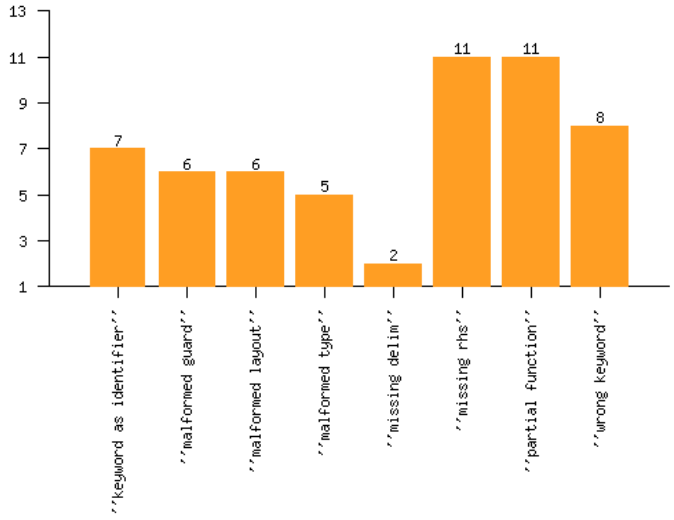

Figure 7.12: Number of parse errors per kind (Week 9)

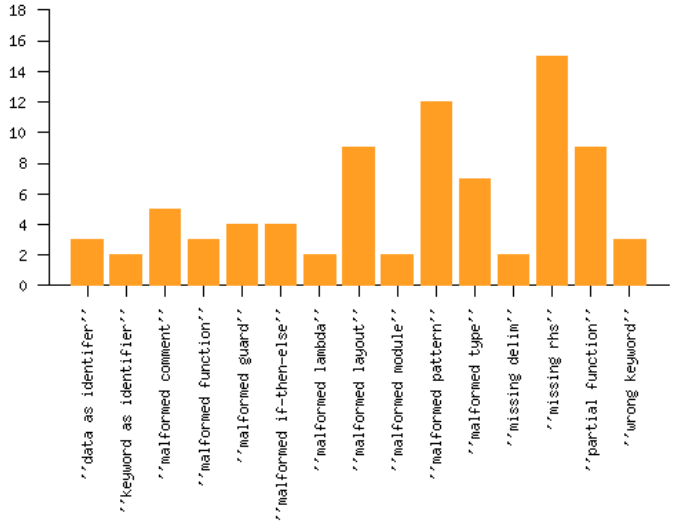

Figure 7.13: Number of parse errors per kind (Week 10)

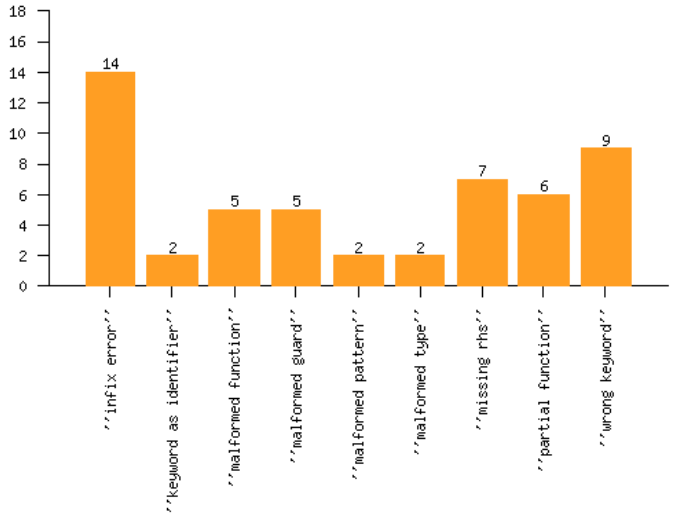

Figure 7.14: Number of parse errors per kind (Week 11)

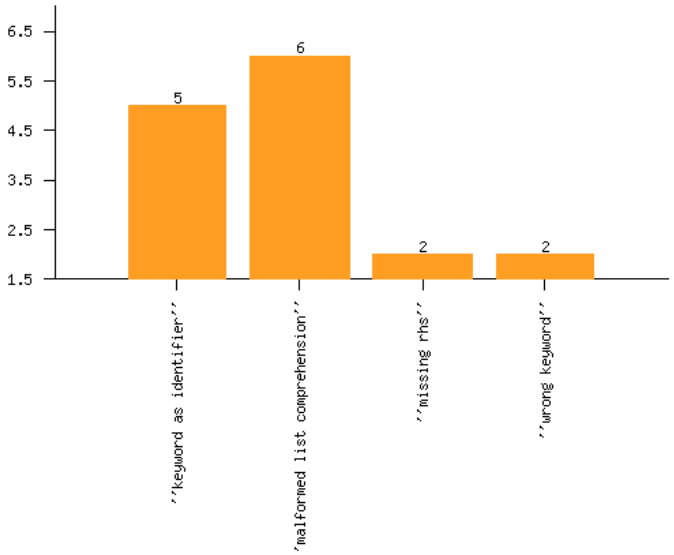

Figure 7.15: Number of parse errors per kind (Week 12)

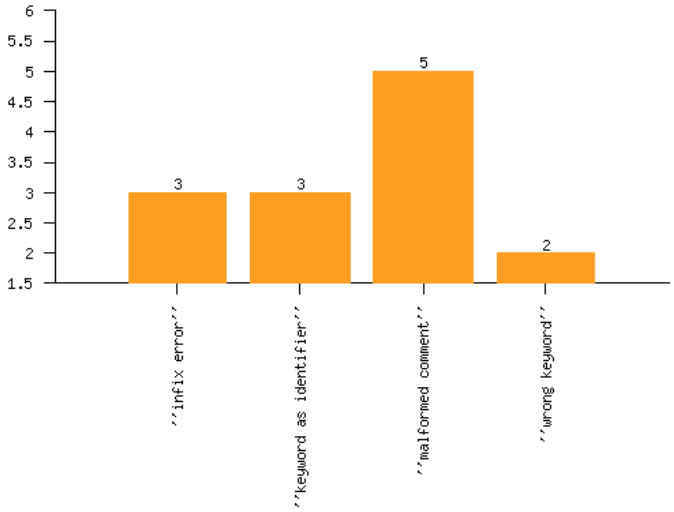

Figure 7.16: Number of parse errors per kind (Week 13

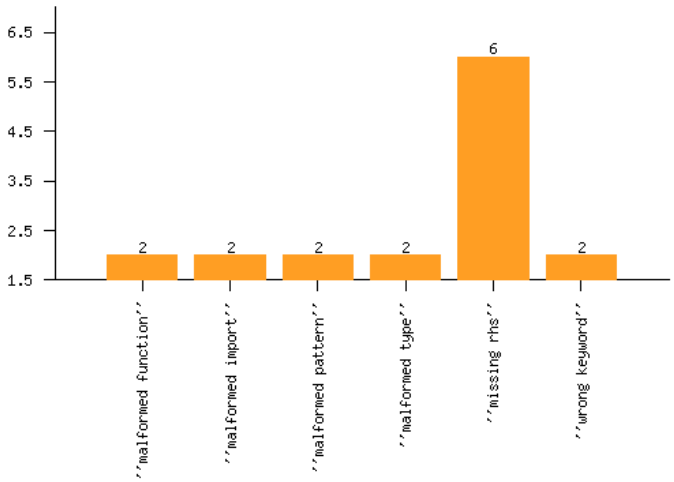

Figure 7.17: Number of parse errors per kind (Week 14)

7.5 Conclusions

Unfortunately we cannot claim that students make fewer parse errors as time progresses. This may be caused by a number of reasons. First of all new language structures are introduced as the course progresses over time. These new structures are structurally different to the ones provided earlier in the course and can therefore cause new users to make errors. Another reason is that a lot of parse errors appear to be caused by students attempting to use Helium to evaluate their partially complete code. This is seen by the large amount of missing right hand side errors in the last analysis. Their goal seems to be to check if the program "compiles up till here".

Chapter 8

Student Data

8.1 Introduction

This chapter focuses on incorporating student data, as presented in section 3.2.1, in the analyses done into errors as presented in chapter 5. As introduced in section 3.2.1, dr. Hage managed to retrieve a lot of data about students who followed the functional programming course. We will use this data to try and see how such external student properties influence his programming style.

The quality of the work (handed in) by students can be influenced by a large number of external factors. Some students have previously followed the course, other have been influenced by imperative programming or an other programming course. This may influence them in their produced amount of errors, or make them produce a different kind of errors. External data may also be an effect of the submitted Helium data, for example the grade received for a practical session is the direct result of the files compiled with Helium, the total grade is a somewhat indirect effect.

We performed a number of analyses in this chapter. The first focuses on the amount of errors / warnings a student make per grade. The second focuses on the amount of errors a student makes per compilation phase per grade.

8.2 Approach

There a number of things to take into account when developing analyses on student data. First of all, loggings can be the result of the work of a number of co-operating students. Prior to 2004-2005 students were allowed

to submit the exercises presented in the practical sessions in pairs. This means our architecture needs to take into account how to handle multiple students. We therefore added a list of student data to our logging data structure.

Not knowing which student actually created this data has a number of implications. How do we handle data which is different for both students? For some research it may be necessary to replicate each logging for both individual student. For other studies it may be required to investigate how to combine both students 'values' to a single one.

The focus in this thesis is the results from the 2004-2005 course, in which students worked alone. This means we can generally ignore these considerations. This may mean however that our analyses can not be easily applied to data from other years.

8.3 Claim

We would like to investigate whether students who get a higher grade program differently than those who get lower grades. We expect to be able to prove this claim by investigating measurements of good coding by differentiating loggings by students grades. Our working hypothesis is that students who receive a better grade write better code.

8.4 Research

8.4.1 Amount of errors and warnings

We will start by looking at whether a students grade influences the amount of errors / warnings he / she makes, as making a lot of errors / warnings is seen as an indication of poor programming. We expect students with favorable grades to make fewer errors and their code to throw fewer warnings. To study this we will focus on the grade which students received for the complete course.

In the design of this analysis we will have to take into account the fact that a logging can be done by multiple students. We can easily accomplish this by taking the average of the grades of students who were in a group which compiled a certain source. We then group loggings by this average grade. The groups in the 2005 course only consisted of a single student however. We incorporated this feature to ensure our analysis will work on loggings from other years as well.

We have to take into account the fact that we do not have a complete list of grades for each student. To achieve that we built in a filter into our new grouping. We write an *selectGrpAnalysis* to remove these students from the list of loggings. We achieve this by filtering out any *Nothing* values.

Next we have to take into account the fact that we may not even have *studentData* for all students. We remove those in our grouping, since grouping on non-existent values does not make any sense. These are most likely students who did not actually take the course, people who just test-drove Helium at the university machines to make use of its nice error messages when compiling Haskell code.

Last we take into account the fact that grades aren't always integer values. Grouping per exact number does not really give us a clear graphical overview, so we decided to group the grades based on their rounded grades.

This leads to the following grouping function:

```
groupPerGradeRounded :: AnalysisKH [Logging] [Logging]
groupPerGradeRounded = groupAnalysis
  (λx →
      ("Rounded grade",f x))
      (λxs → groupBy (λa b → (f a) ≡ (f b)) (sortBy (comparing f) xs)
  )
   <o> mapAnalysis (selectGrpAnalysis "available marks" g)
  where f :: Logging → Int
    f = (round ∘ avg ∘ marks)
    g log = (length (marks log)) ≡ (length (studentData log))
        ∧ ((length (studentData log) > 0))
marks xs = map fromJust
  (((filter ((≢) Nothing)) ∘ (map courseGrade) ∘ students ∘ courseData) xs)
avg :: [Float] → Float
avg xs = (sum xs) / (toEnum (length xs) :: Float)
```

We are able to use this analysis in combination with the analyses we have encountered in the previous sections, namely the *getWarnings* and the *getErrors* analysis. This leads to the following analyses:

```
errorsPerGrade = "errors" <?> divideAna <o> f
  where f = err <&> setSizeAnalysis
        <o> groupPerGradeRounded
    err = sumAnalysis <o>
      mapAnalysis (length <$$> getErrors)
warningsPerGrade = "warnings" <?> divideAna <o> f
  where f = warnings <&> setSizeAnalysis
```

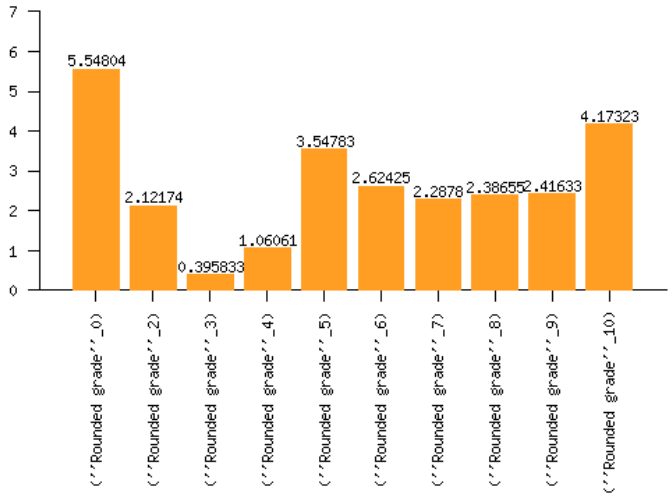

Figure 8.1: Average number of warnings per compile per rounded grade

```
    <○> groupPerGradeRounded
warnings = sumAnalysis <○>
  mapAnalysis (length <$$> getWarnings)
```

These analyses rely on the *divideAna* and *setSizeAnalysis* which are defined as follows:

```
setSizeAnalysis :: (DescriptiveKey key) ⇒ AnalysisFK key [a] Int
setSizeAnalysis = basicAnalysis "size of set" length

divideAna :: (DescriptiveKey key, Real a, Real b, Fractional c) ⇒
  AnalysisFK key (a, b) c
divideAna = basicAnalysis "dividedBy"
  (λ(x, y) → (realToFrac x) / (realToFrac y))
```

In short, the analysis works by tupling the amount of errors / warnings with the original amount of loggings. Next we divide the ammount of errors / warning, which we perform per rounded grade. The results of these analyses can be seen in figures 8.1 and 8.2.

We hoped to be able to see a clear differences in the amount of errors students make who get higher grades, unfortunately this is not the case.

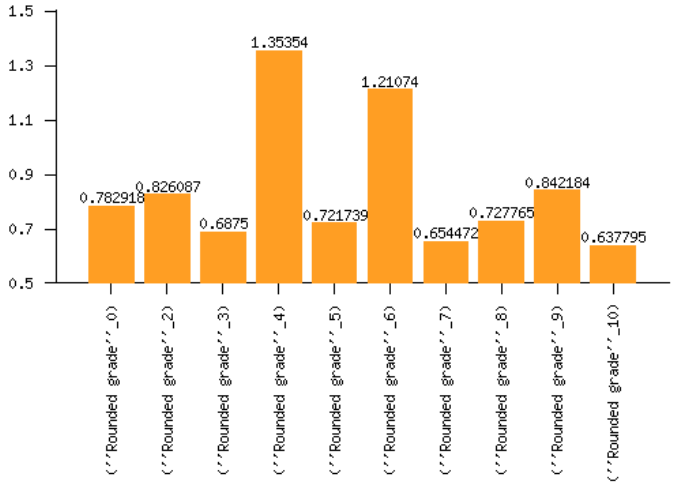

Figure 8.2: Average number of errors per compile per rounded grade

8.4.2 Amount of errors by type

As we failed to indicate a direct correlation between grade and the amount of errors a student produces we will attempt to show one between errors generated in a certain and the amount a student makes. *Do students with high grades make fewer errors of a certain kind than other students?*

To investigate this will require a different type of analysis. We will need to create an analysis which retrieves the amount of errors which is thrown per phase with the total amount of loggings which we retrieved per phase. This results in an analysis that looks like this:

```
relErrorTypesPerGrade' :: AnalysisKH [Logging] ([(Phase, Int)], Int)
relErrorTypesPerGrade' = getErrorTypes
    <&> logsPerGrades <o> groupPerGradeRounded
```

It returns a list of phases with the amount of errors encountered during that phase tupled with the total amount of loggings by students with a certain grade.

We get this list of phases per error by retrieving the loggings which ended in a certain phase. Next we retrieve all errors from the log count them and group the errors by their phase.

```
getErrorTypes = mapAnalysis' f <o> mapPerPhase
  where f = (λx → (ph x, length x))
            <$$> concatAnalysis <o> repliZip
            <o> mapAnalysis' (filterEmpty <o> errid)
```

```
    errid = getErrors <&> id
    ph = phase ∘ snd ∘ head
    filterEmpty = filterAnalysis (λ(x, _) → length x > 0)
mapPerPhase :: AnalysisKH [Logging] [[Logging]]
mapPerPhase = basicAnalysis "per phase"
                (
                  (groupBy (λx y → (phase x) ≡ (phase y))) ∘
                  (sortBy (comparing phase))
                )
```

The definition of the functions *concatAnalysis*, *repliZip* and *mapAnalysis'* can be found in appendix A.3

The *logsPerGrades* analysis simply retrieves the number of errors per grade.

```
logsPerGrades = setSizeAnalysis <∘> groupPerGradeRounded
```

All we need to do now is to divide the amount of errors by the total amount of loggings, which is a bit more complicated than normal, due to the type of *relErrorTypesPerGrade'*. We will have to divide the number of errors, which is stored per phase per grade, by the total amount of loggings, which is stored only once per grade.

```
relErrorTypesPerGrade :: AnalysisKH [Logging] [(Phase, Float)]
relErrorTypesPerGrade = f <$$> relErrorTypesPerGrade'
  where f (xs, len) = map
          (λ(a, b) → (a, (realToFrac b) / (realToFrac len))) xs
```

Note the type of this function, which is required for representing results as a stacked bar chart. The results of this function can be seen in figure 8.3. Unfortunately we do not see any clear patterns in them either.

8.5 Conclusion

Unfortunately we cannot conclude that students who get a better grade for the course make fewer errors. A possible cause may be the fact that student who get higher grades are also those who make use of more complex language structures. Unfortunately we cannot analyse this with the current version of helium, since we have no way to include data only available to the Helium parser.

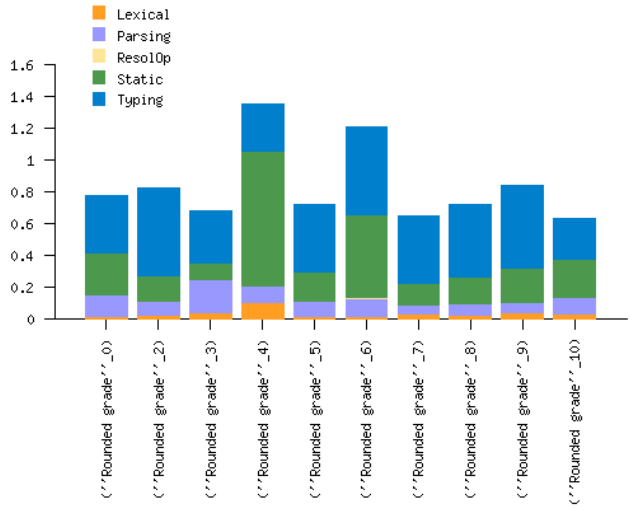

Figure 8.3: Number of errors per compile per rounded grade per phase

Chapter 9

Reflection

Unfortunately we cannot go without spending a few words about the downsides of the approach we followed during our project.

9.1 Approach

Even though we have shown that working on our data in hindsight is possible, there are some severe issues. These are generally caused by the fact that it was impossible for us to envision which data is wanted while the data was initially gathered.

For example, all student data was retrieved by reconstructing data from numerous sources. First of all the student administration, but also the original submission system used by students. The individual grades per course were retrieved by asking the original student assistants who graded the practical assignment for the grades they had sent to the course teacher over five years ago. This will be practically impossible for courses which were taught in the years before 2005. A large amount of the student assistants who aided in the grading of the course are no longer working or studying at the Utrecht University. Getting into contact with them may be hard. Even the teachers of these courses may no longer be available, making the collection of data even harder.

Another issue is that it is hard to write about the analysis that has been developed. There is currently no way of automatically generating data from within a written document. It is also hard to maintain an up to date view on the data being presented, since the code which generated an image may have changed since the time an image was generated. A possibility is to use LhsToTex to run haskell code which would generate the required figures,

but this would require running every analysis again when a document is recompiled. Since even just parsing all data can take a few minutes, we chose not to implement this thesis this way.

9.2 Validity

There are some concerns about the validity and correctness of our data. Working with over 8000 log entries from the functional programming course of 2005 may seem like a substantial database. Unfortunately this does not always hold.

For example, if we would study the difference between the way male and female students create code, we are confronted with the fact that in 2005 only 10 female students enrolled in the course. Another example is the parse errors presented in chapter 7. During week 12 of 2005 only 29 parse errors were encountered. While making claims about this data only 3 loggings by a single student, without changing the actual source, is able to create over ten percent of the errors.

This issue is exaggerated by the fact that the current architecture of Neon does not allow us to focus on different years in a single analysis. Even if we would be able to incorporate data we would still face a number of issues. These are the same issues as described in section 9.1.

Overall we can claim that while looking at 'global' figures the data may seem valid. While zooming into any specific errors it is very hard to claim that the data is valid. This thoroughly undermines the conclusions drawn from the data within this report.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

Doing research into errors and warnings presented by the Helium compiler is possible, by using the Neon framework. We showed a number of examples of questions which can be asked about the errors and warnings encountered by students, like "Do students who get a higher grade make fewer errors?". We have shown how to approach developing analyses which help to answer such questions and created such analyses to help answer the questions presented in our examples.

We contributed to the Neon system in a number of ways. First of all we have developed a system for interpreting the messages which are shown by the compiler at compilation time. We added external (student and course) data to our system provided a mechanism for linking this to individual students. The data available to the Neon system was extended to include the actual messages (errors and warnings) presented to the students. We adapted Neon to be a new development cycle, based on `ghci`, allowing analyses to be developed and performed much faster.

Analyses can be made much more concise by incorporating external data. External data can be used to filter unwanted loggings and indicate different kinds of students, over which research can be performed per group. Our additions to the Neon system to develop interesting analyses which study errors and warnings which resulted from the compilation of Helium source files, we have illustrated how to use them. We indicate how to present the data resulting from performing such analyses using the Neon framework. Developing analyses allowed us to investigate a number of recurring issues, which have been addressed for future analyses developers to take into account. We discussed a number of limitations to our approach, pri-

marily caused by the limited size of the dataset on which we performed our analyses.

10.2 Future work

The Neon project is open ended, there are a lot of possible additional features to the Neon toolkit. Some of these features would be technical, improving the analyses performed by neon, others would be instrumental.

For example, Neon is meant to generate reports. Analyses are hardly interesting without any way to present them. Currently we need to write analyses within Haskell, using the combinators. This is very inconvenient for large structures. For example, this thesis was written in Latex, using Lhs2Tex. We presented the actual code of the analyses next to their results. If we would like to create this within the current Neon architecture we would have to write all text, we would like to mix with our presented data, within Haskell source files. Modifying the thesis would mean recompiling Neon, and re-running every analysis, hardly a convenient process. A way to solve this problem is to write a templating engine, like lhs2Tex, which allows the user to indicate the results of which analysis he wants in his TeX source, where the templating systems would fill in the Neon figures.

A possible extension to the framework would be the implementation of 'more' parts of Helium, for example, the parser. This would allow us to to analyse the actual parse tree as created by the Helium compiler. A downside would be that since Helium differs from year to year, we have to invent some way to deal with that.

An option to the issues which are presented by our relatively small dataset may be presented in performing research over multiple years. We can envision generic timing notions as "first week of the course" on which we can perform groupings. We would have to extend Neon in such a way that it can handle multiple types of Loggings, provided by different years, simultaneously.

Appendix A

Technical

A.1 Data structures

The data structures which were presented by Van Keeken for storing loggings had to be adopted to our new data. The data had to be adapted to include information about students, the "student data", and data about errors and warnings, which are retrieved by parsing the compiler output. We have included the updated structures since they are required to be able to read the analyses presented in this report.

```
data Logging = Log {
                    username         :: Username,
                    phase            :: Phase,
                    heliumVersion    :: HeliumVersion,
                    logPath          :: Path,
                    logDate          :: CalendarTime,
                    staticErrorCodes :: [StaticErrorId],
                    extraLoggingInfo :: ExtraLoggingInfo,
                    compilerOut      :: CompilerOut,
                    courseData       :: CourseData
                  }
  deriving (Show, Eq, Typeable)
```

The logging data type is largely the same as presented in the previous Neon version. We have added new fields to the Logging record, namely *compilerOut* and *courseData*. The types of these fields are found below.

```
data CourseData = CourseData {
    students :: [StudentData],
    grade :: Maybe Float,
```

```
    parseErrorData :: Maybe String
  }
    deriving (Show, Eq, Typeable, Ord)
```

The *CourseData* structure stores the students which were part of the group submitting which submitted this exercise together. We have tried to find a grade which belongs to the practical sessions to which this logging belongs. This is a maybe since we are not always able to obtain a relevant grade. An instance in which it would not be possible to obtain such a grade is when students compiled a source after all deadlines for practical sessions which involved Helium have passed.

```
data StudentData = StudentData {
    attempt :: Maybe Int,
    startDate :: Maybe CalendarTime,
    studentType :: Maybe StudentType,
    courseGrade :: Maybe Float,
    gender :: Maybe String,
    ident :: String,
    imp :: Maybe Float,
    groupId :: String
  }
    deriving (Show, Eq, Typeable)
data StudentType = CKIB | INCA | WISK | NAST
    deriving (Show, Eq, Read, Typeable, Ord)
```

The *StudentData* structure stores the data which is discussed in section 3.2.1. Optional fields, or field which we have not always been able to retrieve are represented as a Maybe. *attempt* refers to how many attempts the students has previously done to pass the course. *startDate* represents the date at which a student enrolled in his bachelor program. *studentType* represents his / her bachelor program. *courseGrade* represents the grade a student received for the complete course. *gender* stores gender. *ident* stores a unique id, this id can exist in multiple groups. *imp* is the grade for the imperative programming course. *groupId* stores a unique group identifier; it should match the *username* from the *Logging*.

```
type CompilerOut = (Version, [CompiledFile])
type CompiledFile = (File, [CompiledPhase])
type CompiledPhase = (Phase, [Error])
type Version = String
  -- Error = (Position, Kind, Message)
type Error = (Position, String, String)
  -- Position = (Line, Character)
```

type *Position* = (*Int*, *Int*)
type *File* = *String*

The CompilerOut structure exists to store the compiler output. The *CompilerOut* type consists of a *Version* which indicates the exact version of the Helium used to compile the source, and a list of *CompiledFile*, which may list the source file and, optionally, some included modules. *CompiledFile* reports the Filename of the compiled file and lists the errors and warnings per phase in a list of *CompiledPhase*. The *Phase* indicates one of the termination phases, which are also matched to the output messages. The *Error* type contains a message thrown by the compiler. A message reports a position within the source file and a message reported.

A.2 New and extended combinators

Neon's primary power has always been a combinators library for expressing analyses. During our development of the new analyses we encountered in this report we noted the need for a few new combinators. For combinators previously developed by Van Keeken we refer to his report [HK07].

Function application The original Neon only allows us to apply a certain analysis to the other. This allows us to chain a number of analyses using the <o> operator. We felt this was a somewhat limited approach however, urging us to develop the new <$$> combinator which allows us to apply arbitrary functions to any analysis.

infixl 4 <$$>
(<$$>) :: (*DescriptiveKey key*) ⇒
  (*b* → *c*) → *Analysis key a key b* → *Analysis key a key c*
f <$$> *ana* = (*basicAnalysis*_ *f*) <o> *ana*

Analysis information A downside of the <$$> operator is the lack of applying a suitable description to a given analysis. We have therefore chosen to introduce a new function for easy application of analysis information. The <? > combinator allows us to define a description for any previously performed basicAnalysis.

infixl 2 <?>
(<?>) :: *String* → *AnalysisKH a b* → *AnalysisKH a b*
a <?> *b* = *basicAnalysis a id* <o> *b*

Tuple lifting The <&> combinator allows us to fork an analysis into a tuple. This allows us to apply analysis on each of the values of the tuple, while maintaining coherence with the other value. The combinator has one problem, namely that it is hard to apply existing analyses to an already analysis that generates tuples. To augment this shortcoming we defined the <&&> operator.

```
(<&&>) :: AnalysisKH a a2 →
  AnalysisKH b b2 →
  AnalysisKH (a, b) (a2, b2)
x <&&> y = λinput →
  f (x $ (fromT fst input)) (y $ (fromT snd input))
  where f = zipWith
    (λ(a, b) (c, d) →
      ((Statistics.DescriptiveAnalysis.combine a c), (b, d))
    )
fromT f input = zip (map fst input) (map (f ∘ snd) input)
```

Silent tupling The <&> is a convenient operator, which has a few downsides. One of the most important downsides is that its key generation function does not allows clearly indicate what actually occurred during an analysis. To complement the <&> combinator we have introduced the &> and < & operators which suppress the alterations to the key type from respectively the first or the second analysis in the tuple.

```
zipByAnas f = zipWith (λ(a, b) (c, d) → ((f a c), (b, d)))
(&>), (< &) :: AnalysisKH a a1 →
  AnalysisKH a a2 →
  AnalysisKH a (a1, a2)
x&> y = λinput → (zipByAnas (flip const)) (x input) (y input)
x < & y = λinput → (zipByAnas const) (x input) (y input)
```

A.3 New common functions

A large number of functions are regularly encountered in this thesis. This chapter lists their definitions and a short explanation of their use.

A.3.1 Thesis.Research.Common

filterAnalysis

```
filterAnalysis :: DescriptiveKey key ⇒ (b → Bool) → AnalysisFK key b b
filterAnalysis _ [ ] = [ ]
filterAnalysis f (x : xs) | f (snd x) = x : filterAnalysis f xs
    | otherwise = filterAnalysis f xs
```

Filters certain values from the input given predicate $b \rightarrow Bool$.

concatAnalysis

```
concatAnalysis :: DescriptiveKey a ⇒ AnalysisFK a [[b]] [b]
concatAnalysis = basicAnalysis_ concat
```

Undoes a nested mapping, while maintaining the key.

expandAnalysis

```
expandAnalysis :: (DescriptiveKey key) ⇒ Analysis key [a] key a
expandAnalysis = concatMap expand
  where expand (key, values) = map (λx → (key, x)) values
```

Undoes any mapping, by replicating the key generated by the input analysis over the new values.

idAggregate

```
idAggregate :: forall a key ∘ (DescriptiveKey key) ⇒ Analysis key [a] key [a]
idAggregate = aggregateAnalysis "joined" concat
```

Aggregate analysis which does nothing, but makes it possible to 'forget' mappings.

joinByKey

```
joinByKey :: Ord key ⇒ Analysis key a key [a]
joinByKey xs = map flat $ groupBy (λx y → (≡) (fst x) (fst y)) $ ord xs
  where flat (x : [ ]) = (fst x, [snd x])
```

```
flat xs = ((fst (f xs)), (map snd xs))
ord = sortBy (λx y → compare (fst x) (fst y))
f [] = error "fout in joinKey"
f (x : _) = x
```

Creates a list for each unique key value.

topN

```
topN n = ("top " ++ show n) <?> (take n ∘ reverse) <∘> orderAna f
  where f xs ys = compare (length xs) (length ys)
```

Selects the *n* most frequent values.

superGroupByOrd

```
superGroupByOrd :: DescriptiveKey key ⇒
  ([a] → String) → (a → a → Ordering) → Analysis key a key [a]
superGroupByOrd keyfun f xs = map comp $ groupBy f' (orderAna f xs)
  where comp xs = (keyfun' (map fst xs) (map snd xs), (map snd xs))
    keyfun' keys vals = Statistics.DescriptiveAnalysis.combine
      (groupKey $ Left (keyfun vals)) (head keys)
    f' a b = case f (snd a) (snd b) of
      EQ → True
      _ → False
```

Analogues to regular *mapAnalysis*, except transforming the type of the original analysis, adding an extra depth.

orderAna

```
orderAna :: (a → a → Ordering) → Analysis key a key a
orderAna f = sortBy (λx y → f (snd x) (snd y))
```

Required to perform groupings on values which do not follow the regular *Ord* ordering.

A.3.2 Thesis.Research.TupledCommon

divideAna

```
divideAna :: (DescriptiveKey key, Real a, Real b, Fractional c) ⇒
  AnalysisFK key (a, b) c
```

```
divideAna = basicAnalysis "dividedBy"
  (λ(x, y) → (realToFrac x) / (realToFrac y))
```

Analysis for dividing values in tuples.

repliZip

```
repliZip :: DescriptiveKey key ⇒ AnalysisFK key [([a], b)] [[(a, b)]]
repliZip = basicAnalysis "zipped" $ map (λ(xs, y) → zip xs (repeat y))
```

Replicates the right value in a tuple and zips is with the left value.

A.3.3 Thesis.Research.ParseErrors

expand'

```
expand' = joinByKey <○> expandAnalysis <○> expandAnalysis
```

Generally used to transfer an analysis of a list of errors to an analysis of errors.

getMessages

```
getMessages :: AnalysisKH Logging [Error]
getMessages = basicAnalysis "Errors" (getErrorsFilterFile (const True)) <○> getCompilerOutput
```

Gets all messages outputted by the compiler.

getErrors

```
filterWarnings = basicAnalysis "nonWarnings" (filter (λ(_, a, _) → a ≢ "Warning"))
getErrors :: AnalysisKH Logging [Error]
getErrors = filterWarnings <○> getMessages
```

Gets all messages which are errors.

getWarnings

```
getWarnings :: AnalysisKH Logging [Error]
getWarnings = "Warnings" <?> (filter (λ(_, a, _) → a ≡ "Warning")) <$$> getMessages
```

All messages which are warnings.

List of Figures

Bibliography

[Gru] Steve Grubb. ploticus. `http://ploticus.sourceforge.net`.

[Haga] Jurriaan Hage. Helium. `http://www.cs.uu.nl/wiki/bin/view/Center/Helium`.

[Hagb] Jurriaan Hage. Top. `http://www.cs.uu.nl/wiki/Top/WebHome`.

[HK07] Jurriaan Hage and Peter van Keeken. The neon dsel for mining helium programs. Technical Report UU-CS-2007-023, Department of Information and Computing Sciences, Utrecht University, 2007.

[HLvI03] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, August 2003. ACM Press.

[Jad05] Matthew C Jadud. A first look at novice compilation behavior using bluej. *Computer Science Education*, 15(1), 2005.

[Kol] Michael Kolling. Bluej. `http://www.bluej.org/`.

[LHPT94] Paul Lukowicz, Ernst A. Heinz, Lutz Prechelt, and Walter F. Tichy. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28:9–18, 1994.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, Department of Information and Computing Sciences, Utrecht University and Microsoft Research, 2001.

[P+03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[ZW97] Marvin V. Zelkowitz and Dolores Wallace. Experimental validation in software engineering. *Information and Software Technology*, 39:735–743, 1997.