

Thesis Proposal

Analysis of Helium programs obtained through logging

Peter van Keeken, 0264539
Student from the master Software Technology
at the Dept. of Computer Science,
University of Utrecht

- *Accepted Version* -

August 12, 2005

Abstract

Helium is a user-friendly compiler especially designed for teaching and learning the functional programming language Haskell. The compiler provides a feature called *Big Brother*, which logs all programs compiled at Utrecht University within the Faculty of Mathematics and Computer Science. These loggings give an enormous amount of interesting data concerning novice programmers, mostly within the course of functional programming. This thesis proposal is about analyzing data and giving insights in ways to evaluate the Helium compiler. Recently implemented help mechanisms, like hints and type inference scripting can be evaluated for effectiveness. Compilation behavior can be used to give a better understanding of the way students tend to program and the performance of the compiler and its features. A result of the research will be a tool set for analyzing Helium program loggings.

Contents

1	Introduction	2
2	Context	2
2.1	TOP	2
2.2	Haskell	3
2.3	Helium	3
2.4	The Helium logging facility	5
2.5	Related work	5
2.5.1	Similar studies	6
2.5.2	Compilation behavior theory	7
2.5.3	The Helium compiler	8
3	Thesis Project	10
3.1	Research questions	10
3.2	Contribution	11
3.3	Approach - Methods and techniques	11

3.3.1	Presenting the analyses results	12
3.3.2	Notions for analysis	12
3.3.3	Building blocks for analysis	13
3.4	Considerations	14
3.5	Project deliverables	14
3.6	Planning	15
4	Conclusion	15
A	Appendix	17

1 Introduction

For the conclusion of my master study Computer Science at the Utrecht University, I am required to do a thesis project in the field of software technology. I will do my project within the TOP [22] research program within the Software Technology group [24] at the Utrecht University, on the analysis of Helium [1, 9] programs obtained through loggings.

The research will analyze collected programs from (mostly) first year students working in supervised and non-supervised laboratory sessions on the computers of the Faculty of Mathematics and Computer Science in the functional programming course at the university. Helium is a user-friendly compiler especially designed for learning the functional programming language Haskell, which aims at providing clear and precise error messages and warnings, in comparison to other Haskell compilers. The version of Helium running on the network of the faculty, provides a feature called *Big Brother*, which logs all programs being compiled on the network. With data we can analyze the programming behavior of students and compiler responses in great detail. Furthermore we can check how mechanisms implemented for error message improvements, like corrective *hints*, are applied by the programmer and how effective the hints are. Since the Helium compiler [1, 9] is developed by researchers from the Software Technology group of our Institute of Information and Computing Sciences, we can make use of the internal parts of the compiler, like the lexical analyser and other parsing components.

Another component in this project is that by making these analyses parameterizable or tunable we create a tool set which is very useful for generating feedback to a teacher of a functional programming course which uses the Helium compiler or interpreter.

2 Context

This section describes the context of the Helium program logging research. The research will be part of the TOP research program and uses tools and languages common within the Software Technology group [24]. Related work will be discussed in Section 2.5.

2.1 TOP

The TOP project [22] deals with the development of tool and library support for building compilers. These compilers deal with improving user feedback during program analysis. The TOP research group has a lot of experience with type inferencing a higher-order polymorphic functional language [6]. This experience is used for building the type inferencer of the Helium compiler. The compiler supports almost full Haskell 98, the latest Haskell language standard. The first Helium compiler was released in November 2002 and has been a platform for the implementation of new ideas in the field of type inferencing.

2.2 Haskell

Haskell is a pure functional programming language based on the functional programming paradigm [4]. This concept of functional programming has been around for a long time. Some even argue that *lambda calculus* could be considered the first form of a functional programming language, although it was not designed to be executed on a computer. *Lambda calculus* is a model of computation, designed by Alonzo Church in the 1930s provides a formal way to describe function evaluation. During the years several researches implemented the ideas behind lambda calculus, in languages like Lisp (John McCarthy, 1950) and Scheme (Guy L. Steele and Gerald Jay Sussman, around 1980). While not a purely functional programming language, Lisp did introduce most of the features now found in modern functional programming languages. Scheme was a later attempt to simplify and improve on Lisp. The language Haskell was designed in the early 90's as pure functional language to gather many ideas in functional programming research, like polymorphic typing and lazy evaluation. The reader is expected to be familiar with Haskell and type inferencing. Recommended references for these subjects are respectively *The Haskell School of Expression* by Paul Hudak and *Types and Programming Languages* by Benjamin C. Pierce, but information can be found on the Internet as well [4, 19].

There are five flavors available for compiling or interpreting Haskell code. The Haskell Interpreter Hugs [5], is a small, portable Haskell interpreter written in C, which runs on almost any machine. It offers relatively fast compilation of programs and a reasonable execution speed, and is available through an interactive interpreter. Hugs also comes with a simple graphics library. However, being an interpreter, Hugs does not nearly match the run-time performance of, for example, GHC, nhc98, or HBC. Hugs is used for learning the basics of Haskell, although the error messages are not as good compared to the Helium compiler and even bad when compared to GHC.

GHC [15], the Glasgow Haskell Compiler, is an open source compiler and interactive environment, developed at the University of Glasgow. It is a full implementation of Haskell 98 plus a wide variety of extensions and works on several platforms including Windows and most varieties of Unix. GHC has extensive optimization capabilities, including inter-module optimization. Profiling is supported, both by time/allocation and various kinds of heap profiling. GHC is designed to act as a substrate for the research work of others. Downsides of GHC as compiler are its size, complexity and memory consumption.

nhc98 [18] is a small, portable, standards-compliant Haskell 98 compiler, aiming at producing small executables that run in small amounts of memory. It produces medium-fast code, and compilation is quite fast. nhc98 comes with tool support for automatic compilation, a foreign language interfacing, heap and time profiling, tracing, and debugging. Some of its advanced kinds of heap profiles are not found in any other Haskell compiler. nhc98 is available for almost all 32-bit Unix-like platforms.

HBI and HBC [2] or also called the Chalmers' Haskell Interpreter and Compiler implements Haskell 98, as well as some extensions. It is written by Lennart Augustsson, and based on the classic LML compiler by Augustsson and Johnsson. The interpreter can also load code compiled with HBC. There has been no official release for the last few years.

Helium [1, 9] is a functional programming language, compiler and interpreter designed especially for teaching Haskell, developed at the Software Technology group [24] at the Utrecht University. Quality of the error messages has been the main concern both in the choice of the language features and in the implementation of the compiler. Helium will be discussed in more detail in Section 2.3

2.3 Helium

Helium [1, 9] is a compiler designed for learning Haskell. The Helium compiler is being developed within the Software Technology [24] group at the Institute of Information and

Computing Sciences of the Utrecht University in the Netherlands. Quality of the error messages has been the main concern both in the choice of the language features and in the implementation of the compiler. The goal of the Helium compiler is to let students (or anyone) learn functional programming more quickly and with more fun. A major difference with other Haskell compilers is that Helium does not fully support type classes¹.

All Haskell compilers and interpreters are able to generate error messages and warnings while compiling a piece of program code. The aim of the Helium compiler is to guide the developer by the error messages and warnings. When these messages are clear to the developer, the developer will become faster in learning to program and doing his job. In contrast with most compilers, the Helium compiler transforms the Haskell code in four fully separate analysis phases. This characteristic in combination with advanced error analysis techniques makes it possible that the reported error messages and warnings of the Helium compiler thrown per phase are much clearer and easier to resolve, than errors or warnings from other Haskell compilers.

The compiler goes through a Lexical, Syntactic, Static and Type checking analysis phase. *Lexical analysis* is the process of dividing program code strings into logical components, called tokens, based on punctuation and other characters, like brackets. This results in a tokenized sequence representation of the programming code. The Helium compiler generates errors and warnings when inconsistencies are detected based on punctuation.

Syntactic analysis is the process of checking whether the code is well-formed according to the Haskell grammar. This phase results in a parse tree in which the logical structure of the program is presented. In this representation there still may be some detectable inconsistencies. These can be detected in the static checking and type checking phase.

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

Type analysis is the process of checking all elements in the parse tree match with their required type. This phase results in a fully type correct parse tree, which can be safely transformed into low level code. The type inferencer of the Helium compiler has the following properties:

- Precise position information of an error is generated and type synonyms in error messages are preserved.
- The programmer can choose the type inference strategy of his liking, like M [13] and W [3] and other greedy variants, or the unbiased type graph based implementations
- The type graph implementations use several heuristics to decide what is the most likely source of the error.
- The type inferencing process can be influenced by external type directives. With these directives the programmer can develop his domain specific type rules for a combinator library he might be writing. In addition to this he can also specify that his experiences are that certain functions are often mixed up. As a result, a compiler may give a hint about alternative functions that fit in the context. The type directives rules are automatically checked for soundness, and a programmer does not have to be familiar with the process of type inferencing as it currently takes place within the compiler. These type directives will be discussed in more detail in Section 2.5.3.

When all phases are passed successfully the program can be executed. It is still possible that errors occur in this runtime 'phase'. Some, but not all, of these errors could theoretically have been detected by an extra 'runtime' analysis. This runtime analysis would be the process of checking if the runtime behavior of the program will result in a

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

legal output. This would not be the case when one is taking the head of an empty list or uses of a integer out of range. The Helium compiler does not provide any functionality to prevent these kind of errors during compilation, since they are hard to locate and is mainly the responsibility of the programmer to avoid these logic errors. But when these errors occur during executing, proper error messages are generated.

A special version of the Helium compiler is developed for storing all compiled program files on a network and the compilation result. This feature is called *Big Brother* and is as such part of the compiler but is not equipped with any checking or analysis facilities. This feature will be discussed in Section 2.4

2.4 The Helium logging facility

Almost all programs being compiled on the network of the faculty of Computer Science are centrally logged with the *Big Brother* feature². For each user, his program is stored sequentially in time during development with an uniquely identifying time stamp and user identification. Every timestamped log consists of the program file undergoing development and the compilation result. Since January 2005 the logger also stores imported program files (e.g. libraries).

The compilation results consists of the following attributes:

- A string identifying the (anonymized) programmer
- The time stamp of compilation
- The filename being compiled
- The version, date and time of the Helium compiler build
- The compilation phase in which the compilation process ended

If the compilation process ended in the static phase, the logger also notes the error message code(s) returned by the compiler. The explanation of these error message codes are described in Appendix A.

All the programs have been anonymized by removing all references to the user, as well as all the comments in the code. To keep the relation between programs from the programmer intact, the user identification string is replaced with an anonymized unique string, linking the program loggings for a student, but without a reference to his or her identity.

There are small issues concerning the use of the results of the logging facility, these will be discussed in Section 3.4

The Helium compiler, with the *Big Brother* feature is are ready put to use in two functional programming courses. In the academic year of 2002 - 2003, this delivered 30,000 loggings from 120 students. In the academic year of 2003 and 2004, this delivered 23,000 loggings from 142 students.

2.5 Related work

This section discusses related work, published in the field of the Helium compiler development and quantitative logging analysis. First an overview will be given of similar studies performed on other (non-functional) programming languages. Then related notions to compilation behavior and quantitative analysis of collected programs will be discussed. Furthermore we will discuss relevant work regarding the Helium compiler and the functionalities, build especially for improving error messages.

²The programmer can turn this feature off by a using a compiler parameter.

2.5.1 Similar studies

Similar research has been done for the imperative languages Cobol [14], Ditrان, a Fortran variant [17], PL/I [25] and the object oriented language Java [16] with the aid of BlueJ [10], an interactive development environment [12]. Although none of these languages are functional, these studies give an interesting insight in the possible analysis on program loggings. It is interesting to note that there is a gap of about twenty years between the first [14, 17, 25] and the recent study [10]. This is probably due to the fact that the early languages were using highly centralized computing facilities, like mainframes. Therefore all programs were centrally processed, making it easy to gather logging information. Until the research with BlueJ and Helium, there does not appear any relevant qualitative study but only qualitative focusing mainly on the cognitive side of the programming process.

Ditrان

The language *Ditrان*, a variant of the language Fortran, was developed for diagnostic and educational purposes at the University of Madison, Wisconsin in 1967. The Ditrان compiler was evaluated by Moulton and Muller in 1967 by making performance records (loggings) [17]. The aim of this research was to evaluate the technical and educational aspects of the language. The evaluation defines 21 error categories typically made by a programmer. Moulton and Muller present overall statistics like: total amount of programs run, compilation time, statement distribution, compilation and execution time error distributions. Of the 5,158 analyzed programs from 234 students (average length of 38 statements), 1859 had compilation errors with an average of 3.8 per program. Although the paper does mention the value of analyzing the complete history of a particular programmer, this information is not available due to the procedure for submitting runs.

PL/I

Marvin Zelkowitz discusses the PLUM system, a diagnostic PL/I compiler developed at the University of Maryland in 1975 [25]. The system stores data from four sequential phases: translation (compilation), execution, post-execution and the language evaluation. 4,583 collected programs are analyzed by program size using a histogram. Furthermore a selection of 5,672 runs is analyzed, representing all runs by students in a programming course. The data collection is analyzed for the effect of following a prior course in structured programming. The analysis is done by examining the frequency of used statements. The students that did follow the course structured programming can be characterized by using 6% to 12% more comments, and fewer 'GOTO' statements. By using these characteristics the paper shows that it is also possible to identify students who did or did not have a prior structured programming course by using a clustering algorithm.

Furthermore Zelkowitz analyzes the program complexity, by measuring statement complexity versus program size; the interactive usage, by scanning for special interactive commands; the error analysis, by counting the programs which caused errors during compilation and execution time, and analyzing the termination responses, if known. Zelkowitz does not present any user specific analyses.

Cobol

Litecky and Davis categorize and analyze the frequency of error messages in the language Cobol [14] in 1976. First Litecky and Davis formulate an error classification scheme of 132 errors by analyzing 1,000 runs from 50 students studying Cobol. Applying this scheme to a second group of 1,400 runs from 37 students (1,777 errors) they conclude that most errors, about 80% are caused by 20% of the error classes. Although there is a truth in

this, this Pareto principle is a bit manipulated, since they freely choose their error classes. Litecky and Davis also analyze the error-proneness of errors by taking into account the number of times the related language element is used. They conclude that only four out all errors are significantly present in the novice programs. Since these errors are not vital to the basic structure of Cobol, the related elements should be changed or avoided in future program design. From comparing the error diagnosis of the Cobol compiler with a human diagnosis, the authors conclude that a stunning 80% of all diagnoses are inaccurate. An overall conclusion is that (related) compilers can be greatly improved by use of the collected statistics and by using mechanisms like automatic error correction and providing hints to the developer.

Java

M. Jadud presents a very similar research for the object oriented language Java [16] applied recently at the University of Kent in 2003 and 2004 [10]. Programming loggings are made by use of the interactive novice programming environment BlueJ [12]. Jadud's research not only aims at understanding program loggings as aspect of the programming process, but also aims at improving the programmer's behavior. With this improving or shaping the programmer's behavior, this research positions itself as quantitative as well as qualitative study. Since this research is still ongoing, there is no final conclusion drawn for most of the qualitative aspects of programming. Whether shaping the programming behavior is desirable or how it is best achieved, stays yet unanswered.

Jadud presents two kind of analysis in his work. The first aims at the whole student population, looking at the types and distribution of encountered errors, the time between compilation, the time between compilation based on the result of the previous run (categorized as 'syntactic correct' or 'not syntactic correct') and the time between compilations as function of the error type. In the latter case the paper concludes that for the three most common classes of error encountered by students, very little time (about 30 seconds) is needed and a few characters are changed to fix the problems. The second category of analysis aims at the individual characteristics, studying the average lab session behavior of students. Jadud concludes that that there are three kinds of behaviors: normal, with an average of 10 compilations per session³; deviant, with a normal average except for one session; different, significantly more or less compilations.

Although the paper presents a very similar research, which can be generalized for our research as well, it does differ in some aspects. E.g. the language analyzed is object oriented and not functional; there is no categorization of errors (a program is regarded syntactically correct or not); by design BlueJ only presents one error at a time, unlike Helium does; BlueJ is a environment for novices, where Helium is a compiler for novices, possibly used with an interactive interpreter.

2.5.2 Compilation behavior theory

From reading the previous section it is clear that there are many ways, that program loggings can be analyzed. The analysis can range from simple statistics to more advanced method taking the complete history of compilation events into account. Unfortunately there is not much material available focusing on the complete compilation behavior. From the research done with Java [16] and BlueJ [10] an additional deliverable [11] describes possible analysis, notions and techniques useful for analyzing compilation behavior of (novice) programmers. Major research questions of Jadud are: what is the reason of a compilation action; what is the resulting difference, the so-called *compilation* Δ , being the events and actions that take place between two compilations; what is the influence

³A session took one hour, and was one time a week.

of an environment on a programming behavior. In this paper Jadud also introduces the notions: *solution trajectory*, the evolution of a student’s work toward a particular solution; *compilation history*, a sequence of time-stamped snapshots of code at various points between creation and compilation.

The compilation history is just one aspect of a solution trajectory, the solution trajectory is far more complex since it concerns everything that goes on in the mind of the programmer. Furthermore the paper describes three kinds of analysis, *trivial* (mean, median, average, etc.), *textual* analysis and *structural* analysis. For the textual analysis the paper proposes techniques similar to the edit distance as used for the *hint* mechanism implemented in the lexical and syntactical phase in the Helium compiler [9, 8]. Here the edit distance is expressed in the number of edit operations (insert, update or delete) to transform from one string to another. But also tools like `diff`, `comm` are proposed [20]. For the structural analysis Jadud presents similar techniques to the edit distance but based operations like insert, update or delete on program element nodes, operating on a parse tree like representation of a program. Here for he uses theories of K. Zhang et al. [26] and D. Shasha et al. [23].

Another aspect of compilation behavior theory is the learning style. A study by Perkins, et al. [21] observes two kinds of learning behavior when learning youngsters to program LOGO, namely *stoppers* and *movers*. Stoppers are children who, when confronted with a problem, just give up. Movers, on the other hand, may get lost, but because they have forward motion, it is possible for them to solve their problems. It could be possible to see a similar behavior in the Helium loggings as well.

2.5.3 The Helium compiler

There are several papers focusing on the development of the Helium compiler at the Software Technology group [9, 8, 6, 7]. In this section we will discuss relevant papers about the Helium compiler and the functionalities offered by the compiler, for improving error messages. A study about type classes directives [6] is not yet implemented in the Helium compiler, but only in TOP, the underlying type inferencing framework of the compiler. We still discuss this topic, since it likely to be part of the Helium compiler in the future and offers ideas for future enhancements for improving error messages.

Learning Haskell

The advantages of using the Helium compiler to learn functional programming in Haskell becomes clear when one compares the error messages of Helium with the error messages from Hugs and GHC [9]. Heeren, Leijen and IJzendoorn [9] show that error messages and warnings from the Helium compiler are better and more understandable than messages and warnings from Hugs or GHC. This is shown for syntactic and type errors as well as errors as they can occur during runtime. The Helium compiler is also equipped with an advanced *hint* mechanism. This mechanism generates possible solution(s) to detected parse problems or (type) inconsistencies. Heeren et al. also present basic statistics from the *Big Brother* feature from a functional programming course in 2002 - 2003, using the Helium compiler. Although these statistics give a good overall insight in the error distribution of novice programmers, a closer analysis of the program loggings is recommended.

Type inference directives

Error messages can be greatly improved by using domain knowledge from e.g. libraries or knowledge about commonly occurring programming misunderstandings. But there are serious disadvantages to incorporate this knowledge into the type inferencer of a compiler. Heeren, Hage and Swierstra present four advanced techniques implemented in the Helium

compiler for improving error messages by influencing the behavior of the constraint-based type inference process [8]. The techniques employ external type inference directives from a separate '.type' file for enhancing the error messages, avoiding in this way the need to change the compiler.

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

A second presented technique is *phasing*. Some language constructs are strongly related to each other, but the abstract syntax tree, generated from those constructs can differ with the view functional programmers have of them. In such cases generated error messages can be unclear and hard to understand. To overcome these scenarios *phasing* offers a mechanism to influence the order in which constraints are globally solved for all type rules. With this better error messages can be reported, that fit with the view of the programmer.

Two other techniques, *siblings functions* and *argument permutation* provide mechanisms for generating useful hints [9]. The *siblings functions* technique provides the ability to relate commonly mistaken functions with a likely alternative, the so-called sibling function. These relations can be stated as directives. If the type inferencer runs into an inconsistency with a function mentioned among the directives, its sibling function(s) is tried as fix for the inconsistency. If the sibling function solves the typing error, then the compiler will generate a hint along with the error, referring to the sibling function as possible solution. The fourth technique is *argument permutation*. If reordering the arguments of a function solves a type inconsistency, then this is provided as a hint along with the error message. These techniques obviously show a big improvement for error messages, and provide a flexible framework specifying error messages. Heeren et al. also warns for misleading hints. To avoid this the writers propose to evaluate the Helium program loggings.

Type class directives

Heeren and Hage [6] describe an extension to the directive framework focusing on improving error messages in presence of Haskell 98 type classes. In their writing, Heeren et al. propose four directives, as well as specialized type rules (as described in Heeren et al. [8]) as addition to Haskell compilers, for improving type error messages. The four directives can be implemented in the Helium compiler, to provide better error messages when using overloading with the Helium compiler.

The first directive is the *never* directive, which excludes a single type from a type class. This is the exact opposite of an instance declaration, and limits the open-world character of the type class. Similar to this directive is the *close* directive. The *close* directive prohibits new instances for a type class. With this the compiler has the advantage of knowing all instances. Knowing the full set of instances offers two optimizations for the type inferencing process. In the rare case of an empty type class, every function that requires some instance of that class, can than be rejected immediately if detected. For the case that a type class has only one member, a predicate can be improved to that specific member. The same reasoning applies, if a type variable belongs to multiple type classes and the set of shared instances is empty or a singleton. The third directive, the *disjoint* directive, specifies that the intersection of two type classes should be empty. The fourth directive, the *default*, helps to disambiguate in case overloading cannot be resolved. This directive refines the ad hoc default declarations supported by Haskell. With each of

the first three directives, a specialized error message can be formulated using attributes, to provide extra context, about the detected inconsistency.

Algorithms and Heuristics

The Helium compiler is also equipped with compiler parameters for choosing the kind of type inferencing heuristic or algorithm, in such a way that fits best with the programming style or kind of error [7]. Heeren, Hage and Swierstra [7] describe the type inferencing process in which constraints are generated, ordered and solved, using a constraint tree. A constraint tree is generated by use of bottom-up type inference rules. Locating the source(s) of an error in such a constraint tree depends on the way the constraints are ordered and solved. Ordering is done by defining a tree walk on the constraint tree. This can be a standard preorder, postorder or more experimental ones such as a right-to-left tree walk. For solving constraints the Helium compiler provides several algorithms, greedy ones, which can act similar to algorithms like W [3] and M [13] and more global analysis of a type graph. When a type inconsistency is detected in the type graph, several heuristics are applied to find the most likely source of the error. This latter algorithm is more complex, but can result in a better error report with a hint, providing a possible fix for the problem.

3 Thesis Project

It is obvious that analyzing the Helium program loggings more closely, can reveal a lot of interesting information regarding novice programmers as well as compiler response. This section discusses the details regarding the research as part of a thesis project.

3.1 Research questions

Basic statistics about the logged programs is available by simply analyzing and counting certain aspects in the overall log file. But there is a lot more to explore and to discover. Interesting facts are also hidden in the program loggings themselves, e.g. information about the way novices tend to program and details about the performance of the Helium compiler, regarding the effectiveness of error messages and hints. The analysis of program loggings to retrieve this information clearly forms a challenging target for our research. With this information we can quantitatively evaluate the Helium compiler and test whether the compiler fulfills the statement of being a user-friendly compiler.

There are several aspects that can be evaluated. For our research we will focus on the following research aspects with their related analyses:

- **Program characteristics:** How complex do (novice) functional programmers write their programs? What is the typical program (or module) size (e.g. in number of lines or (oplevel) functions)? How many function parameters are typically used? What is the number of elements on the right hand side of functions? What kind of errors are (mostly) reported (in number or percentage)? This can be analysed per error category or other division. How well are the program documented? (For this we need the original, unanonymized data set) How many warnings concerning a missing type signature is generated?
- **Programming style:** In what way do novice programmers tend to develop their programs? What is the time between compilations? How fast to program grow in length? What is the typical *compilation style*, by incremental extension of a parseable program or by applying a 'type all code, debug all errors' method? Does this differ during the hours of supervised lab sessions or not. Is it possible to detect notions of *programming sessions* as parts of the process of programming? How do

the mentioned program characteristics change over time, during such a programming session? Do warnings about a missing type signature, dissolve fast? Is there a relation between the development style and the ability to recover from errors?

- **Error response:** How do novice programmers respond to errors? How long does it take them to solve a (certain kind of) error? How long does it take to move to a next phase in the compilation process? How many changes are (approximately) applied to fix an error? How are errors distributed over time during programming sessions and does this differ if the subject is being supervised as part of a lab session? Especially interesting are type errors since they occur more often and are usually harder to fix for novice programmers [9]. How well do type directives help in improving type errors? In what cases are misleading or unclear error messages provided by the compiler? What is the result if other algorithms or heuristics would be applied?
- **Hint mechanism:** How often is a hint provided and with which (kind of) error (expressed as number or percentage) is or is no hint provided? Can we identify classes or errors where hints would be useful? Does the programmer apply the hint or not? In what cases is a misleading hint given, like e.g. with the *arguments permutation* technique [8]? Which underlying technique is responsible for the misleading hint? How could other techniques improve the *hint* mechanism?

These questions form a starting point for our research. But there can be more questions, possibly directed by the results from this first set of questions. This calls for a clear way to express the questions, preferably in a framework or standard query language. A basic framework is given in Section 3.3, where we discuss an abstract mechanism for answering research questions. This forms merely a startingpoint since it is not the intention of this research to develop a complete query language specification for this.

There are more aspects that can be of interest like compiler performance or language evaluation. But because of the limitations of time we limit ourself to these four research aspects and their related research questions. If time permits we will look at the learning style of novice programmers using the Helium compilers. Research question related to this are:

- **Learning style:** What kind of *learning style* is common among novice functional programming students, stoppers or movers [21]. How does it differ from the learning style in imperative or object oriented languages like Java [10]. Does the learning style differ during the hours of the lab sessions or not. Are session characteristics much different between the functional, object oriented, or imperative paradigm?

3.2 Contribution

Evaluating the program loggings should give us some insight in the way novice programmers use the Helium compiler for functional programming, and how they respond to the notions inherent to this language, like hints and error messages. This is valuable information not only for a teacher of functional programming courses, but also for researcher in the functional programming and development community. The tool set from the analyses can be used for future compiler developments as well as a framework for analyzing new programming concepts.

3.3 Approach - Methods and techniques

To be able to answer the research questions stated in Section 3.1, the program logs need to be analyzed more thoroughly than by just counting and selecting special occurrences. Automated processing of the logging data is an obvious necessity.

This section will discuss how the analyses can be achieved, without going too much into details regarding the actual program development. First we will describe how the

analysis results can be presented using basic statistics. Next to this we discuss notions that provide a context to large sets of program loggings. Furthermore we will describe building blocks as concept for answering (sub)questions related to logging analysis. We close with a subsection regarding the implementation details and the tools and language(s) that can be best used, when implementing the analyses.

3.3.1 Presenting the analyses results

Most of the research questions mentioned with the aspect of *program characteristics* in Section 3.1, are values that can be computed by taking one logging into account. These measures are relatively straight forward to calculate and are independent of their surrounding loggings in the data set. This is e.g. the case for program characteristic of the program length. This is not the case for all measures. Measures like e.g. the error distribution over time in a session, are more complex to calculate and are depend on multiple loggings. This first group can be presented by using basic statistic measures, like average, minimum, maximum (using percentages or exact numbers) and the measures for distribution like median, mode and standard deviation. For the latter group more advances techniques are necessary. The data set should first be structured into relevant groups of loggings to be able to calculate a value for such a group. The notion regarding these structures will be introduced in Section 3.3.2. To present the result of these more complex analyses, also more elaborate statistical methods, like histograms and charts, can then be used, as addition to the basic statistic measures.

3.3.2 Notions for analysis

To be able to talk about the large set of loggings more specifically we introduce the following notions. These notions function as basic structure to give the logging data a context within the process of programming. With this we can apply an analysis more specifically to a certain group of loggings. These notions refer to the abstract term of *programming session* as mentioned in Section 3.1 about the research aspects.

Basic logging relation or *coherence*, is the notion that a (single) logging is likely to be similar to its previous or following loggings. So program loggings can have certain coherence in content toward each other. This coherence forms a (reflexive, not necessarily transitive) relation between program loggings. The factor of coherence can be tested by parameterizable characteristics, like e.g. the maximum number of different lines between two program loggings. More about the implementation of this notion is described in Section 3.3.3.

A *session* is a sequential set of programs in which all programs are logged within a certain time span. This element of time can be expressed as absolute start and ending time for the loggings or as a relative maximum time difference between all sequential program loggings in the set.

A *block* is a sequential set of programs in which all programs are logged within a certain time span like the notion of session. Besides this the loggings also need to show some coherence in content. It depends on the actual parameters of the basic logging coherence notion, as described previously, how much coherence is required (or difference is allowed) between the loggings.

A *trace* is set of blocks with coherence among the following blocks without necessarily the blocks themselves being directly following each other up in time. This means that other program loggings, blocks or sessions can be present in time between the blocks which are element of the trace.

These notions are graphically displayed in figure 1. In this figure a valid trace can be formed by block 1 and 3 if they show coherence in content within the parameters of this notion.

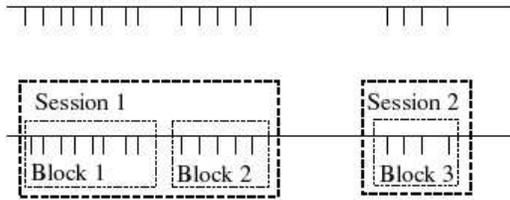


Figure 1: Graphical representation of programming logging context notions

3.3.3 Building blocks for analysis

An analysis can be formulated by specifying a concrete question. Finding an answer to this question in a large data set can be a hard job. Usually a question can be formulated in terms of (more) primitive questions. These primitive questions can also be part of other major research questions. By making building blocks for each (sub)question and relating the building block with each other, we can answer current and future arising questions in a flexible and effective manner. By making the building blocks parameterizable, will allow us to create a tool set for analyzing current and future data sets. Possible building blocks can be categorized in *computational* and *filtering* building blocks.

Computational building blocks are building blocks that retrieve or calculate an actual piece information for one or more program loggings. Since an analysis can apply to one single program logging, a sequence of program loggings, multiple sequences of program loggings or a random set of program loggings, this also holds for computational building blocks. Examples for computational building blocks are the calculation of mean, median, average of certain program characteristics, like program length, function length, or error rates of certain kinds of errors, phase in which the compilation process stopped or growth of program length over time.

Filtering building blocks are building blocks that limit or structure the loggings from a baseline set according to certain characteristics. Examples of these characteristics can be the fact that the selected program(s) have some basic relation or coherence or are part of a session, block or trace. These context notions are described in Section 3.3.2. Filtering building blocks can be combined with computational building blocks, requiring the latter to hold for a certain result. An example of this is a filtering building block which selects all programs for which the compilation process ended in the type inferencing phase.

Implementation

For the actual implementation of the building blocks several tools and techniques can be used. Tools like GNU `diff`, `comm` and `cmp` [20] can be used for the analyzation of text and files. This in combination with regular expressions, tools and languages supporting regular expressions, can form a starting point for prototype implementations. For the overall development of the tool set and the more complex analyses, Haskell is likely choice, since we then can easily reuse parts of the Helium compiler, like the lexical analyser. Comparison on the level of parse trees requires that the programs are parseable in the first place.

The building blocks can be parameterized for variabilities. For example the implementation of the notion of basic logging coherence from Section 3.3.2, can be based on

characteristics like the maximum number of lines of difference between the programs, the number of differences in the top level bindings or simply by matching file names. The notion of session can be based on a maximum time span between the loggings or a begin and end time, to better match the concept of a lab session. The notion of block strongly depends on the notion of basic logging coherence and session. The notion of trace is also based on the notion of basic logging coherence, block and a possible time span maximum between the blocks of a trace.

The combination of the implementation of these building blocks and a (graphical) user interface, to specify and perform certain analyses, form a tool set, also usable for future analyses.

3.4 Considerations

It is our belief that from the program loggings a clear view of the compilation history can be formed. The sequence is only slightly polluted by imported programming files (like libraries) which are sent to the compiler first if a compiled version was not present on the local machine. These 'semi-loggings' are logged close in time just before the logging of the actual developed file. Fortunately they are usually not undergoing active development, and are therefore not often present in the set of logged programs. But since the transfer delay and compilation time of the imported files is unpredictable, analyzing a sequence of loggings should not only consider the direct following program but also a small range of following files.

Until January 2005 the *Big Brother* feature of the Helium compiler only sent the file being compiled. Recently this changed, from January 2005 also all imported program files are stored with the logging, making it easy to recompile and reproduce the compilation result of the logged program. To be able to do this as well for the data set already required, the imported files have to be collected from previous loggings. This process is relatively easy but tedious. For an exact reproduction of the compilation result, the same compiler version should be used as well. This preprocessing of the data sets is part of the thesis project.

A comment on the analyses that will be done, is that the results strictly only form an approximation within the analyzed data set. A reason for this is that not all program compilations are present in our data set. For instance compilations done by a programmer outside the network of the university are not present. Besides this, the program loggings miss a substantial part of the context of the whole process of programming. Still we are sure that an analysis result provides an accurate view and can be used in a broader context since we use such a large data set. But for all statements made, these comments should be kept in mind. This is a common aspect of empirical studies.

3.5 Project deliverables

Like with all researches, this research will be described in a written report. Besides this the research will also deliver a tool set for the analysis of (future) program loggings. Prior to all of this, the logging data will be first preprocessed, to be able to generate the exact compiler report for each logging. With this also most of the used compilers will be provided.

To sum up this thesis project will have the following three major deliverables:

- Preprocessed program loggings and used compilers.
- A thesis report, describing all work that has been done.
- A tool set, for applying analyses for future data sets.

	Action	Duration
1.	Preprocessing	
1.1	Familiarizing with loggings and preprocessing	1 week(s)
1.2	Build compiler versions and basic shell	1 week(s)
2.	Development of building blocks	
2.1	Building blocks for program characteristics	2 week(s)
2.1	Building block for Sequential coherence	2 week(s)
2.2	Building block for Session recognition	1 week(s)
2.3	Building block for Block recognition	2 week(s)
2.4	Building block for Trace recognition	1 week(s)
2.6	Building blocks for computational analyses	
	Programming style, Error response, Hint response	4 week(s)
2.7	Development of (Graphical) User Interface	2 week(s)
3.	Finalization	
3.1	Running analyses and tool set finalization	2 week(s)
3.2	Writing thesis report	2 week(s)

Figure 2: Planning

3.6 Planning

The following planning gives an overview regarding the work that needs to be done. Roughly speaking there are four phases. The preprocessing, the making of the analyses from building blocks, the application of the analyses and writing of the thesis report. Figure 2 shows the tasks needed to be done and the projected time needed to accomplish them. The activities do not necessarily follow each other directly in time.

4 Conclusion

This document describes the research for analyzing Helium program loggings. This analysis can give interesting views on programming behavior of novice programmers, as well as compiler response concerning error, warning and hint messages. This research will evaluate Helium as user-friendly compiler, especially designed for learning Haskell, and providing clear and precise error messages and warnings. A result of this research will be a tool set for the analysis of program loggings.

References

- [1] Daan Leijen Arjan van IJzendoorn and Bastiaan Heeren. The Helium compiler. <http://www.cs.uu.nl/helium>.
- [2] The Chalmers' Haskell Interpreter and Compiler. <http://www.cs.chalmers.se/~augustss/hbc/hbc.html>.
- [3] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.
- [4] The Haskell homepage. <http://haskell.org>.
- [5] The Haskell Interpreter Hugs. <http://www.haskell.org/hugs/>.

- [6] Bastiaan Heeren and Jurriaan Hage. Type class directives. 2005.
- [7] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59 – 80, Cork, September 2003.
- [8] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.
- [9] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.
- [10] Matthew C. Jadud. A first look at novice compilation behavior using BlueJ. 2004.
- [11] Matthew C. Jadud. Naive tools for studying compilation histories. March 2003.
- [12] M. Klling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. Dec 2003.
- [13] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, 1998.
- [14] Charles R. Litecky and Gordon B. Davis. A study of errors, error-proneness, and error diagnosis in Cobol. *Commun. ACM*, 19(1):33–38, 1976.
- [15] S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. <http://www.haskell.org/ghc/>.
- [16] Sun Microsystems. <http://java.sun.com/>.
- [17] P. G. Moulton and M. E. Muller. Ditran - a compiler emphasizing diagnostics. *Commun. ACM*, 10(1):45–52, 1967.
- [18] The nhc98 compiler. <http://www.cs.york.ac.uk/fp/nhc98/>.
- [19] Wikipedia on Type Inferencing. http://en.wikipedia.org/wiki/Type_inference.
- [20] The GNU Operating System. <http://www.gnu.org/>.
- [21] D. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of learning in novice programmers. 1989.
- [22] TOP project website. <http://catamaran.labs.cs.uu.nl/twiki/st/bin/view/Top/WebHome>.
- [23] Dennis Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002.
- [24] Software Technology Website. <http://www.cs.uu.nl/groups/ST>.
- [25] Marvin V. Zelkowitz. Automatic program analysis and evaluation. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 158–163. IEEE Computer Society Press, 1976.
- [26] K. Zhang, J. T. L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937, pages 395–407, Espoo, Finland, 1995. Springer-Verlag, Berlin.

A Appendix

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

Figure 1: Reported error codes in static compilation phase and referring descriptions.