Mining Helium programs with Neon

Jurriaan Hage

Peter van Keeken

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2007-012

www.cs.uu.nl

ISSN: 0924-3275

Abstract

Over the years we have collected a large collection of `Haskell` programs written by students in a first-year functional programming course using the `Helium` compiler. The mining of such a collection is not trivial, especially since the programming was done *in vivo*, and hence largely outside our control. We have developed a sizable library in `Haskell`, called NEON, for computing characteristics of this collection of programs and presenting the results visually. These computations range from simple kinds of analyses like computing the average length of a program per student to determining how long it takes for a programmer to resolve a type error.

1 Introduction and motivation

When the `Helium` compiler for learning `Haskell` was developed in Utrecht [5], a lot of effort was made to improve error messages for novice students. The major innovation of the compiler was to use type graphs and heuristics on such type graphs to improve type error messages. Some of these heuristics were built-in [3], others were in the form of type inference directives that can be specified in special `.type` files that accompany the ordinary `Haskell` sources. This allowed the customization the behaviour of the compiler for classes of expressions, opening up possiblities for supporting domain-specific type-error messages for domain-specific libraries [4].

Although such an innovation seems worthwhile at first glance, its worth in a practical sense can only be established emperically. For this reason a logging facility was added to `Helium` which logs all the programs compiled by a programmer (if he does not explicitly turn it off). This has resulted in a large collection of programs (about 68,000, collected during various incarnations of the functional programming course at Universiteit Utrecht).

In this paper we describe our experiences in mining this huge collection of programs, offer abstractions that turned out to be useful when such is attempted, discuss the Neon library to deal effectively with the implementation of analyses of our collection of loggings, and identify problems we have run into. Many of these problems have to do with a lack of control of the experimental situation. Indeed, we did not actually perform a controlled experiment, but analyzed logged programs after the fact, making it more apt to talk of data mining. The advantage of our set-up is that with the necessary care being taken, we obtain lots of information at little or no cost. This information can help us to improve our compiler, but it can also teach us about how students program, which concepts students use or avoid a lot, where and when they make the most mistakes, but also how long it takes them to correct a mistake detected by the compiler.

To illustrate our work, we have performed a number of analyses of which we give the results in Section 2. In Section 5 on implementation, we show how one of these can be implemented. In Section 3 we discuss concepts we used, taken from the field of descriptive statistics and in Section 4 we discuss concepts that are specifically useful within our domain. In Section 6 we discuss related work, and Section 7 concludes with a discussion and directions for further research.

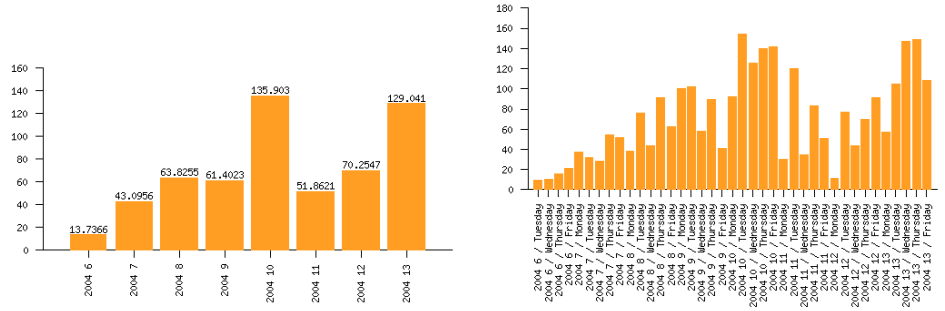


Figure 1: Average module length in terms of lines of sourcefile per week (left) and per day (right) based on the 2003/2004 data set.

2 Examples

The examples in this section serve to illustrate the possibilities of our library when applied to our collection of logged programs. Our main interest with these examples is to show the kind of queries that can be posed to our collection of programs, and not to investigate a specific hypothesis; we leave that to future papers. The master thesis of the second author [11] contains a more thorough description of the analyses including their implementation and an interpretation of the results, with pointers for further study.

Number of lines analysis

To get an impression how the length of compiled programs evolves during the course, we have the computed minimum, average, median and maximum number of lines for the loggings from the course year 2003/2004. Figure 1 (left) gives the average module length for each of the first eight weeks of the course (the final two weeks no loggings were made), as generated by the `ploticus` program [2] to which we fed the computed data. In the histogram of Figure 1(right), we give the same for each day that loggings were made. The fact that week 10 had a deadline should not come as a surprise now. In Section 6.1 of [11] similar pictures are given for individual students, while Section 6.3 contains a more refined analysis showing how the number lines of comments evolves over the course, as compared to the number lines containing actual code.

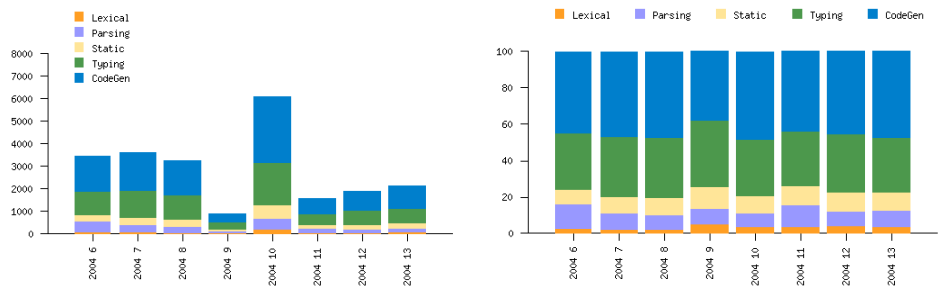


Figure 2: Absolute and relative number of compiles per phase, given per week from 2003/2004.

Phase analysis

The `Helium` compiler may terminate in one of a number compiler phases (due to a programming error of some kind), or it may terminate due to an internal error (of the compiler), or it results in a correct compilation and generates code (the CodeGen phase). The four most interesting compiler error phases are Lexical, Parsing, Static (simple static errors such as undefined or multiply defined identifiers) and Typing (for a type error).

In this analysis we compute for each week during the course in the year 2003/2004 and for each of the five phases, the number of compiles terminating in that phase. The result of this computation is given in Figure 2(left). To be able to compare their relative values, we computed the ratio between each of these numbers and the total number for that week, and obtained the results displayed on the right. In both figures, the x-axis displays the weeks (6–13 of the year 2004) in which the loggings were made. The y-axis of the figure on the left gives absolute logging counts, the y-axis on the right gives the ratios (both cumulatively). The latter shows that over the eight course weeks, these ratios hardly change, except for a noticeable dip in the ratio of parse errors, setting in after the first week. However, towards the end of the course, this ratio, surprisingly maybe, increases again. What the reasons may be for this phenomenon is not easily determined and needs further investigation. For example, it may be due to the fact that difficult syntax is introduced towards the end of the course, but other factors could be involved as well. In Section 5, we show how Figure 2(left) can be computed.

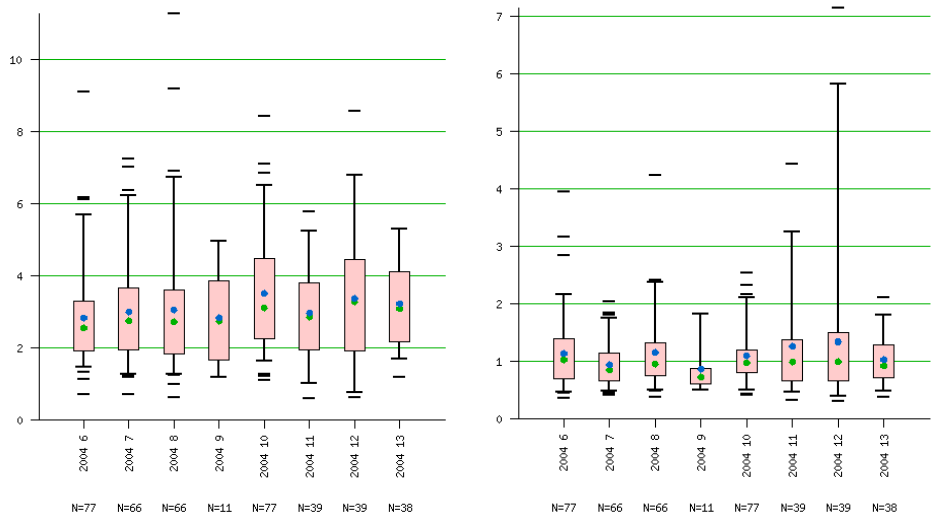


Figure 3: Average (left) and median (right) compilation intervals (in minutes) for all students, given per week with a time coherence of 60 minutes from 2003/2004.

Time between compiles

In this example, we are interested in the spread of in-between compile time (within a programming session). A programming session terminates when no compilation has been made for sixty minutes. We compute the average and median for each student over the entire course. The results can be found in Figure 3, the averages on the left, the medians on the right, given for each week separately. It shows that recompilation times generally range between two and three minutes, and low in-between times tend to occur frequently (which results in the median being lower than the average). Some students take almost ten minutes on average, and another stays below one minute. Note that these values are somewhat influenced by the fact that compiles of imported modules are also counted among the loggings, and typically the in-between time between a compile of a module and one of its imports is only a matter of milliseconds. However, in the class room setting compiling imported modules is not something that occurs very often. We can conclude from the pictures that there are no notable changes in behaviour during the course.

Type error repair analysis

We have performed two related analyses on our loggings to discover how much effort students need to repair a type error. The idea is to look for a type error logging, and then determine how many compiles the student needs to arrive at a correct compile. For each student we compute the average of such values per week, to see whether this changes over time.

For a given student, we first compute sequences of loggings that deal with the same program file, as an approximation of the fact that the sequences deal with the same program: we do not want that a correct compile of another module is viewed as a repair. Then we break each of these sequences into smaller sequences that start with a type error and end in a correct compile. For example, given the following sequence of phases of loggings $[C\ P\ T\ T\ T\ P\ T\ C\ L\ L\ P\ P\ T\ C\ P]$, we obtain two sequences $[T\ T\ T\ P\ T\ C]$ and $[T\ C]$. Then we compute the lengths of these sequences and average these values for each week. The results of the computations for all students in the year 2003/2004 can be found in Figure 4(left). The measured value here is the average number of compiles as just discussed. Alternatively, Figure 4(right) gives the same results, but now we measure the average time needed to solve the error, computed as the difference (in seconds) between the time stamps of the first and last logging in each sequence. In Section 6.5 of [11] similar pictures are given for individual students.

Type hints analysis

One of the features of `Helium` is that some (type) error messages were accompanied by a hint telling the programmer how the problem might be resolved. The first thing one would like to know is how many compiles actually result in error messages that contain a hint. In Figure 5, the ratio of compiles that contain at least one hint (note that every compile may give rise to multiple errors) is given for the year 2003/2004. The ratio is given per week, because we would like to investigate how the ratio evolves over time. As can be seen from the picture, there is a steady decrease in the ratio. There can be various reasons for this: it may be that the hints help students avoid the same mistake later, but it may also be that hints are available for mistakes that are easy to avoid with some practice. In any case, the picture paints a suggestive picture that demands further investigation. For example, we could compute for each type of mistake separately how the ratio evolves over

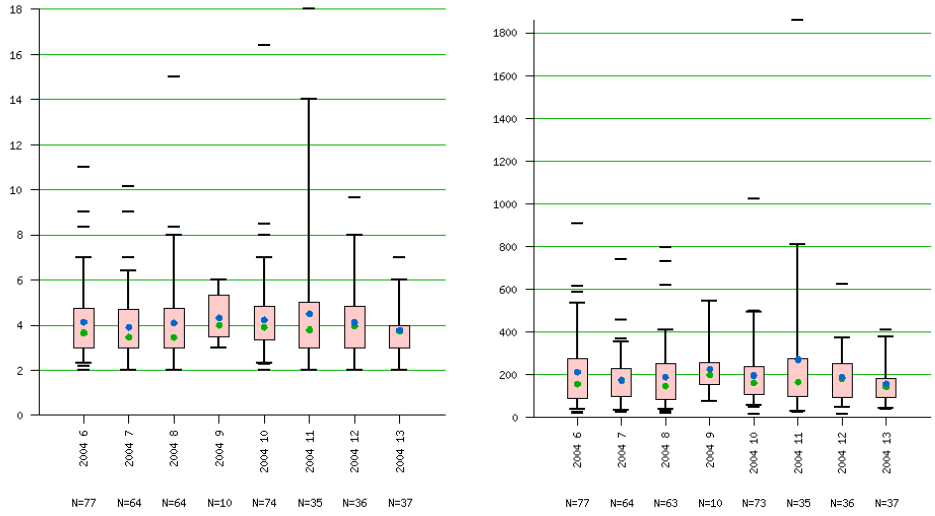


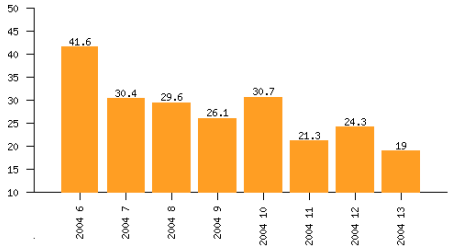
Figure 4: Average number of compiles (left) and average time (right, in seconds, 10 minutes time coherence) needed to repair a type incorrect program for all students, from 2003/2004.

time. Note that in this particular case we have also included a LaTeX table that contains the same information as the picture, also generated by Neon.

3 Concepts from descriptive statistics

Many of the concepts around which our library has been built are not new. They come from the area of descriptive statistics which deals with how to summarize data, either with the goal of showing similarities or by showing how they differ. Specifically, we have sought provisions for dealing with the following issues:

- Group loggings (repeatedly) into groups of related loggings.
- Computing statistical or computational characteristics of the loggings in each group.
- Selecting individual loggings or groups of them based on some computed characteristic.



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

Figure 5: Hint ratio for type incorrect compiles, from 2003/2004, given per week.

- Presenting the results of our analyses in various ways. Essential here is to support ways in which the library can fill in much of the details needed for such presentations automatically.

Since many of these operations are available in database query languages such as `SQL`, a valid question is then why we did not use databases. Currently, our data consists of collections of files, accessed most easily through an ordinary (hierarchical) directory structure, which is not that easily expressed in a relational database schema (but it could be done of course). Also, running a compiler over a program is easier when it is simply part of the filesystem, instead of in a database. Finally, the type of queries we are interesting in computing need a general programming language, and not one suited specifically for databases: we will need to compare programs (in various ways), use regular expressions, and so on.

A major reason for using `Haskell` is that we want easy access to `Helium`, in order to reuse parts of that compiler directly for some of the analyses. For instance, to compute the maximal nesting depth of a `Haskell` program we would like to use the `Helium` parser (itself implemented in `Haskell`), preferrably directly. Also, many of our analyses are built from smaller analyses by means of some form of composition, and this we felt is most easily expressed with higher-order functions.

Taken together, `Haskell` is a natural choice. Given that choice, the option to use a `Haskell` library for interfacing with a database management system is still open, but for now, we kept the number of dependencies on other libraries as low as possible.

4 Coherence

In this section we consider a number of abstractions that we have found useful, essential even, for building analyses on sequences of loggings. In a sense, they can all be mapped back to concepts from Section 3 and the notion of clustering in data mining, but their importance warrants a separate description.

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

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

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

Two subsequent loggings are *time coherent* if they are apart at most k time units for some k, i.e., thirty minutes or 24 hours. The actual choice of k depends very much on situation, so it is a parameter of the definition. We call two subsequent loggings *content coherent* if the contents of the compiled modules are similar (to an extent which is a parameter of the definition). Various instantiations of the term *similar* are likely to be of use: the modules must be exactly the same, the modules differ in at most a single line, the modules have the same name, or the modules differ in at most one top-level definition.

The need for these notions become apparent when one considers the Type Error Repair Analysis discussed in Section 2. During error repair, it might well be that after some attempts the loggee gives up, and goes home.

Two days later, he proceeds with the task, but in the meantime he probably spent only little time on the problem. Thus, it is unlikely that we want to consider these 48 hours as time spent on solving the mistake. When one considers the time to correct an error, one would like to apply these to a subdivision of the loggings, those that are time coherent for a suitable time limit. The notion of content coherence also plays a role here, because when one breaks the sequence of loggings (for a particular loggee) up into subsequences based on time, then such a subsequence may contain loggings that deal with different programming problems. Here the notion of content coherence can be used to distinguish between these different programming tasks, so that a type error in module `A.hs` is not considered "solved" because the loggee compiles an unrelated, correct module `B.hs`.

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

The final concept we introduce is that of a trace. In our situation we will often be interested in all the compiles made by the loggees for a given programming assignment. However, such an assignment usually takes more than a single programming session. Furthermore, the loggee might write and compile other programs during this period, for instance as part of exercise classes being followed. A *trace* refers to a sequence of loggings that deal with a single programming task: it consists of loggings that are content coherent with arbitrarily large look-ahead. In this case it often pays off to

consider modules to be similar if and only if they have the same name or if they have similar contents. Even if someone takes his work home and returns later with a considerably modified version, we can still consider the subsequent compiles part of the same trace as long as he did not also change the filename. Note that obtaining a trace does not mean that we have all the loggings for all the compiles of the program (see our discussion earlier), but having the traces is a good starting point for finding out whether this happens to be the case.

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

5 The Neon library

With a plethora of potentially interesting analyses, it is essential to offer support for implementing these analyses. Therefore, we implemented a library to support building them, which can also export the resulting information to various tools for (graphical) presentation. In this section, we describe the basic types and components of this library and how they can be used to compute one of the examples of Section 2. A more detailed explanation, and the code for the examples of Section 2 that we do not consider here, can be found in Chapter 6 of the master thesis of the second author [11].

An analysis result is represented by a list of key-value pairs, $[(key, value)]$. Here, *value* is the result of the computation and *key* is a description of this value. An analysis then simply maps between two types of this kind:

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

To be able to compose analyses easily, we have chosen to always map a list of pairs to a list of pairs, even if an operation like grouping is involved. For example, suppose we have the following intermediate result: $[(\texttt{"st1"}, \mathit{ls1}), (\texttt{"st2"}, \mathit{ls2})]$, in which *ls1* and *ls2* contain the loggings of student *st1* and *st2* respectively. Suppose the next operation is to group all the loggings for each week together. If *st1* has loggings in week 1 and week 2 and *st2* only in week 2, then the result of this operation would typically be

$[(\texttt{"st1; wk1"}, \mathit{ls11}), (\texttt{"st1; wk2"}, \mathit{ls12}), (\texttt{"st2; wk2"}, \mathit{ls22})]$

Although we have essentially applied two groupings on the original sequence of loggings, this fact is apparent only in the key value.

Although NEON is built on combinators that abstract away from any particular kind of *key*, the library also contains versions for instances of a particular type class, so that the key transformation functions are provided by the instance, once and for all.

Analysis combinators

We shall now describe the primitives and combinators of our library. The primitives derive from the area of descriptive statistics, while the combinators are higher-order functions that build an analysis out of others. We give only the types of the combinators. Their implementation is straightforward (usually one line of code) and can be found Chapter 5 of [11].

The basic operation for calculating a new value from a previously computed value, can be implemented by the *basicAnalysis* primitive. There are slightly different variants available, but a typical one is the following:

$\mathit{basicAnalysis} :: (\mathit{keya} \to \mathit{keyb}) \to (a \to b) \to \mathit{Ana}\ \mathit{keya}\ a\ \mathit{keyb}\ b$

The first function argument specifies how the key values describing the values a are transformed into the *key* values describing the result after performing the analysis. The transformation of a to b is, not surprisingly, handled by the second function argument.

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

$\mathit{countLoggings} = \mathit{basicAnalysis}\ (+\!\!+\texttt{"; Number of loggings"})\ \mathit{length}$

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

To specify a grouping, we define the *groupAnalysis* combinator which has the following type:

$groupAnalysis :: (a \to key1 \to key2) \to ([a] \to [[a]]) \to Ana\ key1\ [a]\ key2\ [a]$

Implementation is slightly different from the *basicAnalysis* since the result of applying the value transformation (second argument) is a list of lists, that is flattened before it is returned as the result of the analysis. In this case the key transformation function is a bit more complicated. To be able to compute a value that describes the outcome, we pass the old key (describing the computations done so far) and an element (in our case the first) of the list. Usually a grouping collects together the loggings that share a given property, say the week in which a logging was collected. The key transformation function can now obtain the week number of the group from its first argument, and reflect this value in the newly computed key.

The group selecting combinator, applies a filter to the groups and has the following form:

$selectGrpAnalysis :: (key1 \to key2) \to (a \to Bool) \to Ana\ key1\ a\ key2\ a$

The *selectGrpAnalysis* function selects only the groups that fulfill the predicate function, and is rather like HAVING in `SQL`. The key transfer function can transform the old key to reflect which filter has taken been applied.

The group aggregator combinator creates an aggregating analysis:

$aggregateGrpAnalysis :: ([a] \to [keya] \to keyb) \to ([a] \to b) \to Ana\ keya\ a\ keyb\ b$

This function aggregates over all the values (in all groups) in the analysis sets, and returns a single aggregated outcome. Again the key can be transformed to reflect this fact.

Although these analysis functions serve well to illustrate the basic ingredients, we have a number of slightly more general versions. The added generality takes the form of a more general key transformation function. More details can be found in Chapter 5 of [11].

To compose two analyses, we can use the $\diamond$ combinator, similar to function composition.

$(\diamond) :: Ana\ keyb\ b\ keyc\ c \to Ana\ keya\ a\ keyb\ b \to Ana\ keya\ a\ keyc\ c$

The $(\times)$ operator tuples two analyses, in case they are to be applied independently to the same input.

$(\times) :: Ana\ ka\ a\ kb1\ b1 \to Ana\ ka\ a\ kb2\ b2 \to Ana\ ka\ a\ (kb1, kb2)\ (b1, b2)$

Other useful combinators like *mapAnalysis*, *splitAnalysis*, *isolateAnalysis*

and • are omitted for reasons of space.

Specializations of the primitive functions

The above described primitives are easy to implement, but not so easy to use. This is mainly due to the flexibility in the choice of the key datatype, and the fact that this type may change during analysis. In practice, changes during an analysis are not likely to be necessary, and additionally, it is not very likely that a programmer would like to use a key datatype other than the ones provided by the library itself, especially since these also make it easier to generate informative pictures for the analysis results.

The first step towards specialization is the *Key* type class, which handles some of the administration of keys, by specifying a start key for a given type of key, and by specifying how keys can be combined into new keys. Implicit in the *Key* class is that the input and output key are of the same type. This already implies a simplification to the earlier functions, by replacing the old *Ana* type with a new one in which the key type is fixed:

type $AnaF\ key\ a\ b = Ana\ key\ a\ key\ b$

The most important type class is *DescriptiveKey* that encapsulates the fixed key transformation function for each primitive analysis on the instance data type. By making a type an instance of this type class, special primitives can be used in which it is unnecessary to specify how the key should be transformed.

For an instance of *DescriptiveKey* we can use the following alternative to compute a grouping analysis:

$$groupAnalysis :: (DescriptiveKey\ key, DataInfo\ b) \Rightarrow \\ (a \to b) \to ([a] \to [[a]]) \to AnaF\ key\ [a]\ [a]$$

The first argument $a \to b$ describes the property on which grouping takes places, while the second tells us how the grouping should be performed. In

$$groupPerPhase = groupAnalysis\ phase\ (groupAllUnder\ phase)$$

we want to collect all loggings in the same phase together, whether they are adjacent in the original sequence or not (this is what *groupAllUnder* does). If we can assume that these are already adjacent, we can use Haskell's *groupBy*.

Presentation issues

Continuing our discussion of *groupAnalysis*, the additional *DataInfo* class encapsulates information necessary to automate the generation of presentations. For example, it encapsulates information to generate titles for the axes of a graphical display, but also deals with the tricky problem of *completion*. Consider for example the *Phase* datatype which represents the phases of the compiler. After performing an analysis, computing some value for each phase, it may happen that one of these phases is not present. For some presentations, we should add these missing phases to the result, and choose a measured value (for example, 0 in the case of integers). This is tiresome, and it would be nice to automate also this aspect of generating presentations as much as possible. Indeed, this is the reason that the function $a \to b$ and the class constraint *DataInfo b* are part of the type of *groupAnalysis* at all.

A lot of the actual work in building the library has been to generate suitable graphical and textual presentations of the analysis results. To be able to do this effortlessly is essential for the economic use of the library and obtaining new results quickly. The library caters for this need by providing means of generating textual data in the form of one and two dimensional tables in LaTeX and `HTML`, and to support (a number of) graphical plots by generating plot files for a tool called `ploticus`. At this moment, we support a variety of bar charts and box plots. From the `ploticus` files we can easily generate Portable Network Graphics files (such as those found in Section 2). More details on can be found in Section 5.2.6 of [11].

An example

As an illustration of how the above primitives and combinators can be used to obtain the results depicted in Section 2, we present here the code for computing Figure 2(left), which contains for each phase the absolute number of compiles. In this analysis we use the specialized forms which use the *KeyHistory* datatype, but note that this is only noticeable from the type signatures.

```
loggingsPerPhase  :: AnaF KeyHistory [Logging] Int
loggingsPerPhase  =  countNumberOfLoggings
                     ◇ groupPerPhase
                     ◇ mainPhasesAnalysis
groupPerPhase = groupAnalysis phase (groupAllUnder phase)
```

```
mainPhasesAnalysis :: AnaF KeyHistory [Logging] [Logging]
mainPhasesAnalysis = basicAnalysis "" (filter ((∈ mainphases).phase))
  where
    mainphases = [Lexical, Parsing, Static, Typing, CodeGen]
loggingsPerPhasePerWeek = loggingsPerPhase ⋄ groupPerWeek
```

To generate Figure 2(left) that displays the number of compiles for each week, we can use the *renderBarChartDynamic* function provided by NEON to generate a `ploticus` file (which can then be used to generate the left hand side of Figure 2(left).

```
phaseResearch :: FilePath → [(KeyHistory, [Logging])] → IO ()
phaseResearch researchdir input = do
  renderBarChartDynamic researchdir (loggingsPerPhasePerWeek input)
  return ()
```

We have omitted the details for the function *groupPerWeek*, but the implementation is straightforward. What is important, is that in the analyses we have implemented these (grouping) functions are used over and over.

6 Related work

The analysis of `Haskell` programs to obtain information about how `Haskell` is used is still very much in its infancy, and as far as we have been able to determine, the same holds for other programming languages. Scouring the Internet we found only a few papers that consider issues related to ours, and relatively scattered throughout time.

Starting in the seventies, a number of studies considered the programming behaviour for various imperative languages: Moulton and Muller [9] evaluated the use of a version of FORTRAN developed especially for education, Zelkowitz [12] traced runs of programs written by students to discover the effects of a structured programming course, and Litecky and Davis [8] considered 1,000 runs from a body of 50 students and classified their mistakes.

A related track of research is the investigation in which way language features influence how easily students learn to program and/or internalize certain aspects of a programming language. These studies usually do not concern themselves with actual feedback provided by programming environments. This type of study goes back to the work of Gannon and Horning [1] in which they compare two syntactically different but similar programming

languages. Also in this case, there does not seem to be much recent activity in this field, and our work can surely contribute to this area, e.g., by examining interference or synergy between languages. For example, do students who know how to program in `Java` make the same kind and number of mistakes as students who do not?

Work was done in the early nineties at Universiteit Twente on teaching the functional programming language Miranda [7] to first-year students. The study was performed empirically by following and interviewing a subset of a group of students enrolled in their first-year functional programming course. The outcomes are of various kinds: they identify problems when learning Miranda, they discovered interferences with a concurrent exposure to an imperative language, and they even went as far to compare problem solving abilities of students who only did the functional programming course with those who did only the imperative programming course.

A more recent study was performed by Jadud by instrumenting BlueJ (a `Java` programming environment) to keep track of the compilation behaviour of students [6]. In his study he determines for various types of errors how often they occur. The similarity with our work is that he also considers compilation behaviour as the subject for analysis. Since imperative languages usually do not have a complex type system, the focus in this work is, like its predecessors from the seventies, on syntactic errors in programming.

A recent study and quite close to "home" was made by Ryder and Thompson [10]. They discuss the application of metrics (defined on `Haskell` programs) to investigate correlations between these metrics and bug fixes made to the program at a later stage. The application of their work is to identify positions in a program that are likely to benefit most from refactoring. Their experimental data consists of two program development histories, based on commits to a CVS repository. The main problem, as they identify themselves, is the fact that they simply do not have enough experimental data to validate their conclusions. Also, they do not have all compiles available, only what was committed, and is thus not suited for the kind of questions we are interested in. We can, however, reuse their metrics and examine correlation with, for example, the final grade for the program or some measure of the effort it took to write the program (the number of compiles for example). It would be very time consuming however, to determine correlation between the metrics and bug fixes, because like Ryder and Thompson we would have to resort to manual inspection to find out which modifications are bug-fixes and which aren't (in this case bug-fix refers to changing a

program that correctly compiles, but that does not fulfill its specification).

7 Discussion and future work

In this paper was have introduced the NEON library we have developed for analyzing a large collection of logged `Helium` programs, based on an understanding of the particular problem area. We have based our library on notions from descriptive statistics, and described the notion of coherence as an important part of the type of queries we would like to pose. We did not provide a coherent study of a particular subject, but have tried to give an idea of the capabilities of the library by means of a number of illustrative examples.

One of the obvious candidates for future work, is to perform a detailed study of a particular hypothesis. Before we actually perform such detailed analyses, a simple, but very useful extension is the use of student characteristics. Although we insist on anonymity, many interesting questions can be posed to our collection based on such characteristics. For example, do non-`Java` programmer make the same kind and the same amount of mistakes as do `Java` programmers? Did the student pass the course in that course instance? Did he work together with somebody else on the program or did he work alone? Did he do the course before? This kind of information can help increase the external validity of results: if groups of students with widely different backgrounds yield comparable results for a query, then it is likelier that these transfer to the programming population as a whole.

The fact that the analysis tool and the compiler are written in the same language, allows us to reuse large parts of the compiler to "easily" compute and compare quite a bit of information about programs between students. Performing diffs between abstract syntax trees of subsequent compiles can tell us a lot about how much students change between compiles, and also whether they tend to stick to solving one problem at the time. Although it will take some work, we think the rewards of taking our work further can be of help in many areas: to improve and determine the quality of exisiting compilers (and related tools), to investigate how students (learn to) program, and even to identify weak spots in the programming language. The logging facility can then be used to examine the effects of measures taken based on this first evaluation. Every compiler should have one!

Acknowledgments We thank Stefan Holdermans, Bastiaan Heeren and Michael Stone for their kind support.

References

[1] J. D. Gannon and J. J. Horning. The impact of language design on the production of reliable software. In *Proceedings of the international conference on Reliable software*, pages 10–22, New York, NY, USA, 1975. ACM Press.

[2] S. Grubb. Ploticus website. `http://ploticus.sourceforge.net`.

[3] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation of Functional Languages – IFL 2006*, volume 4449, pages 199 – 216, Heidelberg, 2007. Springer Verlag.

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

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

[6] M. C. Jadud. A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1):25 – 40, March 2005.

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

[8] C. R. Litecky and G.B. Davis. A study of errors, error-proneness, and error diagnosis in Cobol. *Communications of the ACM*, 19:33 – 38, 1976.

[9] P. G. Moulton and M. E. Muller. Ditran: a compiler emphasizing diagnostics. *Communications of the ACM*, 10:45 – 52, 1967.

[10] C. Ryder and S. Thompson. Software metrics: measuring haskell. In M. van Eekelen, editor, *6th Symposium on Trends in Functional Programming, TFP 2005: Proceedings*, pages 119 – 134, Tallinn, 2005. Institute of Cybernetics.

[11] P. van Keeken. Analyzing Helium programs obtained through logging. `http://www.cs.uu.nl/wiki/Hage/MasterStudents`.

[12] M. V. Zelkowitz. Automatic program analysis and evaluation. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 158 – 163. IEEE Computer Society Press, 1976.