Mining for Helium

Jurriaan Hage

Peter van Keeken

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-047

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 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 [6], a lot of effort was put in 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 [4], 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 [5]. For example, the `Helium` distribution contains a file Prelude.type that specifies, among others, that (:) and (++) are siblings. The compiler uses this information to determine in which situation a type incorrect expression in which (:) is used can be made type correct by replacing (:) with (++).

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, 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 non-trivial analyses of which we give the results in Section 2. In Section 6 on implementation we give some of the code necessary to compute one of these examples.

The paper is structured as follows. We start with a number of examples queries, or, actually, the outcome of these queries, to give an idea of what is possible. We follow up in Section 3 with a discussion of the context of our work, the experimental situation. Here we consider how we have obtained our logging data, how we prepared it for analysis, and what limitations there are to our set-up. Next we describe concepts useful to our work: in Section 4 we discuss those from the field of descriptive statistics and in Section 5 we discuss concepts that are specifically useful within our domain. A discussion of our implementation in the form of a library called NEON can be found in Section 6. In Section 7 we discuss related work, and Section 8 concludes with a discussion and directions for further research.

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 another paper.

Time between compiles

In this first 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 for each student over the entire course. The results can be found in Figure 1. Here, every blue dot stands for a student, and the median is given by the black dot on the right. The picture is generated by a graphical program called `ploticus` [2] to which we fed the computed data. We did the same for the other examples, although our library can also generated `HTML` or LaTeX tables. It shows that recompilation times generally range between two and three minutes, but some students take almost ten minutes on average, and another stays below one minute.

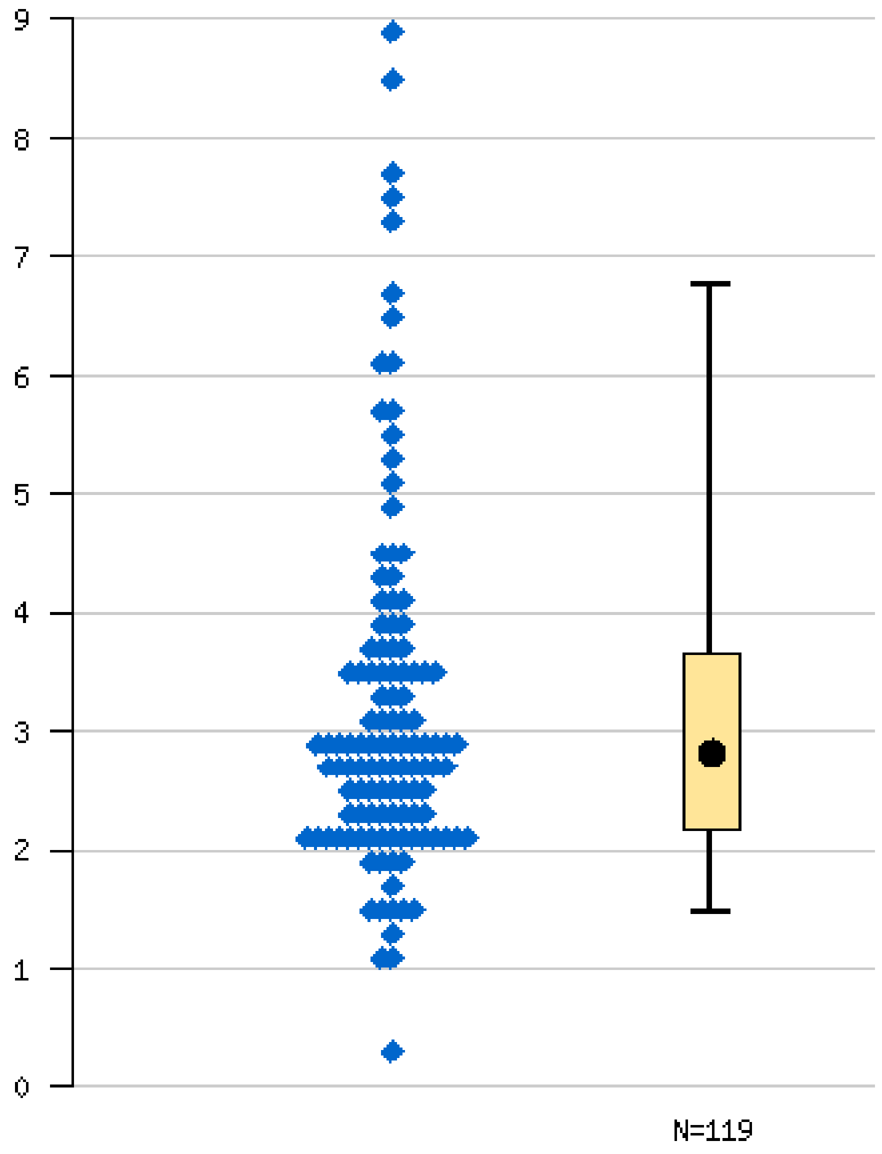


Figure 1: A plot of the average in-between compile time in minutes per student

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 2004/2005. The display to the left of Figure 2 gives the information for each of the first eight weeks of the course (the final two weeks no loggings were made). In this box plot, the green dot is the median and the blue one the average. The box contains fifty percent of the loggings, while the approximate minimum and maximum are given by the lower and upper horizontal lines. They are approximate, because in this type of picture outliers are ignored, and it is possible that the range given by the vertical black line holds only 95 percent of the total number of loggings for that week.

In the histogram on the right of Figure 2, we give only the average number of lines for the compiled modules, but now for each day that loggings were made.

Phase analysis

The `Helium` compiler may terminate in one of five 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). For the compiler five phases, we use Lexical, Parsing, ResolOp (denoting an ambiguity in the resolution of operator fixities and associativities), Static (simple static errors such as undefined or multiply defined identifiers) and Typing (for a type error).

In this first analysis we compute for each week during the course in the year 2002/2003 (29,000 loggings) and for each of the seven phases, the number of compiles terminating in that phase. The result of this computation is given in the left-hand side of Figure 3. To be able to compare their relative values, we computed also the ratio between each of these numbers and the total number of loggings for that week, and obtained the results displayed in the right-hand side of Figure 3. The generation of the data itself took about an hour on a Linux Pentium 4 PC. In Section 6, we give some of the code for computing the data.

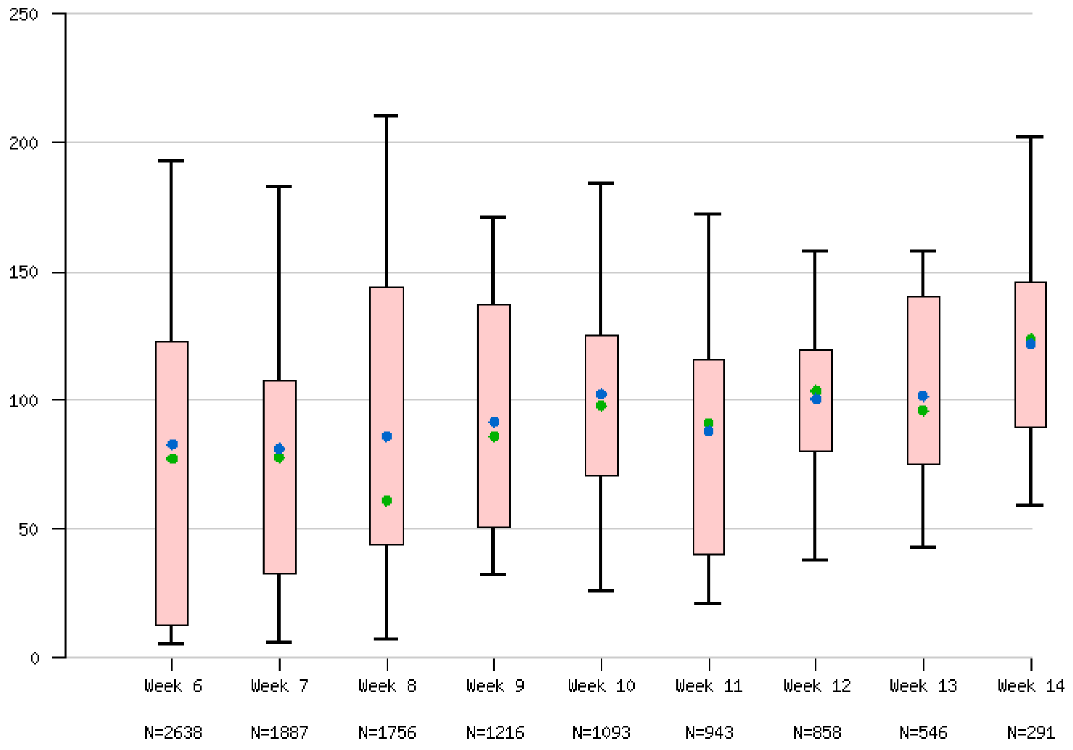
In both left-hand and right-hand side, the x-axis displays the combinations of year (2003) and week (11–17) in which the loggings were made. The y-axis for the left-hand side gives logging counts, the y-axis for the right-hand side gives the ratios cumulatively. The right-hand side shows that over the seven 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 established 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.

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 two students can be found in Figure 4. The x-axis again denotes the week within the year 2003, and the measured value is the average number of compiles as just discussed.

A similar study deals with the time needed to solve the type error. In this case we first break up the logging sequence (for a given student) into sessions (subsequent loggings are no more than



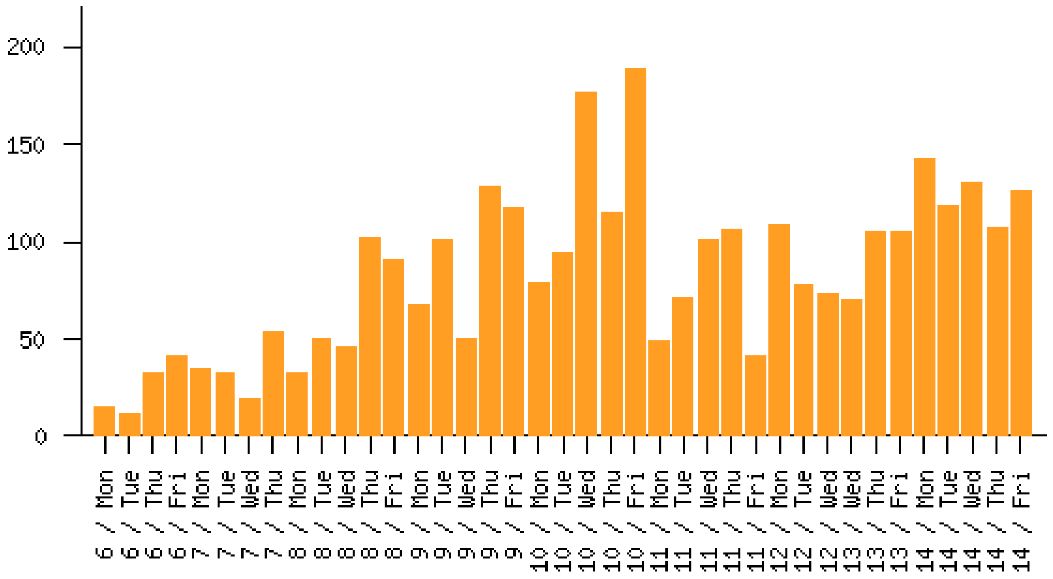


Figure 2: Module length in lines of code given per week (left) or by day (right)

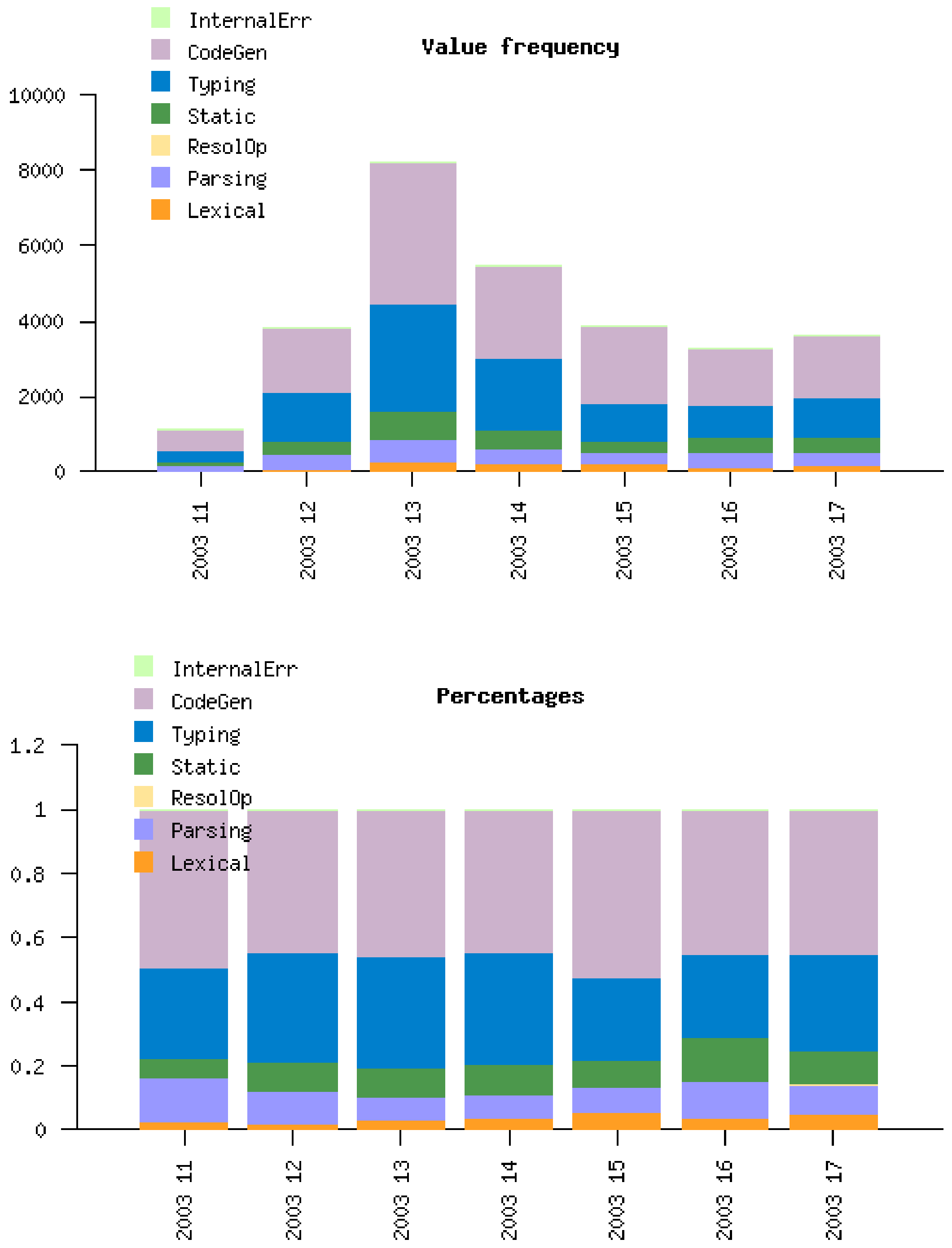
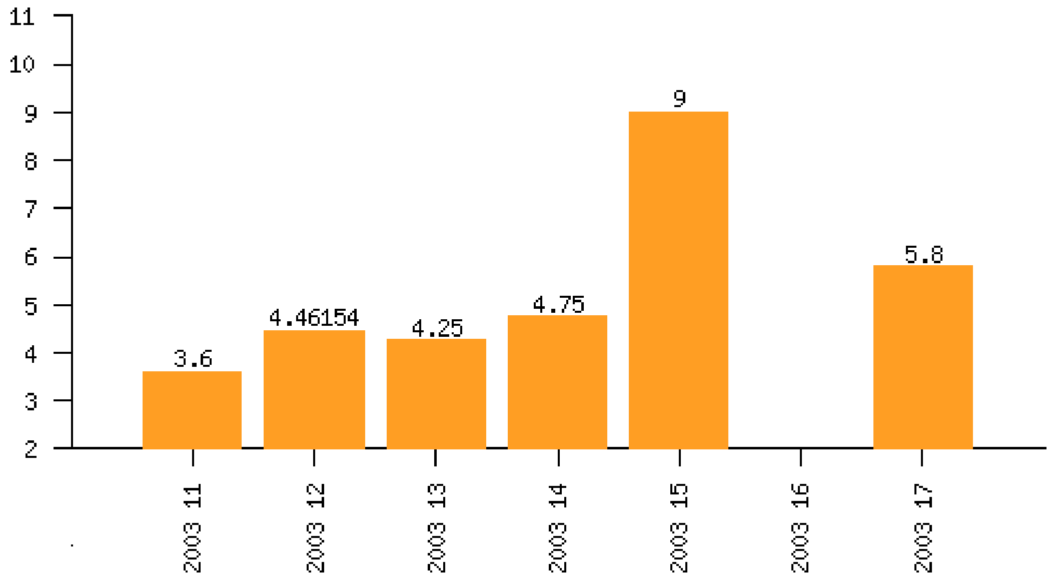


Figure 3: The absolute number and the ratio of compiles per phase, given per week



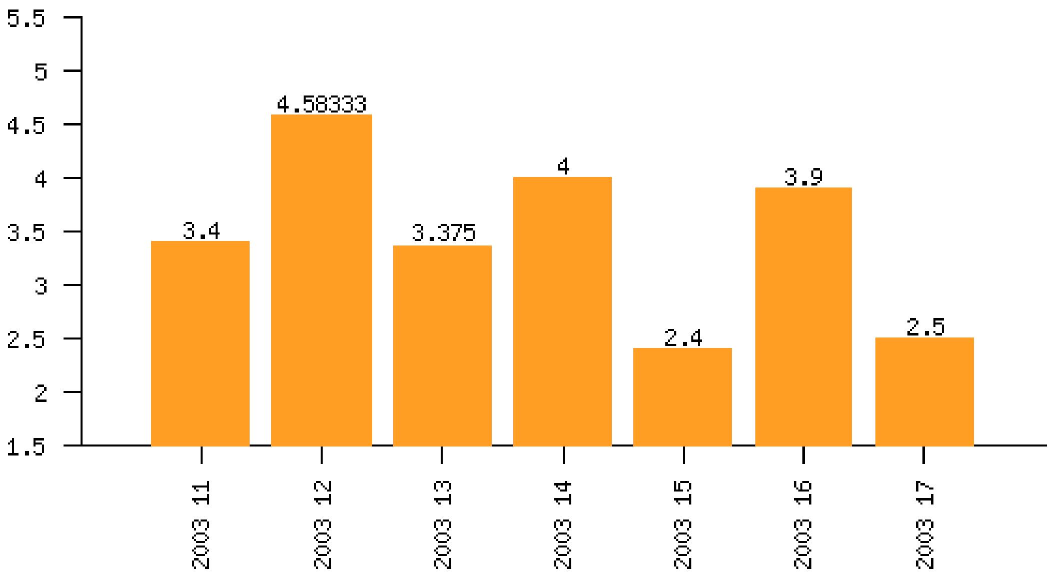


Figure 4: Average number of compiles needed to solve a type error (for two students)

thirty minutes apart) and then again we consider only sequences that deal with the same file. Then we compute the same thing as before, but now we are not interested in the number of compiles, but in the time it takes to resolve the mistake. Note that we split up the logging sequences into sessions first to avoid the situation that a student fixes a problem two days later. This would certainly muddy the waters, because he is probably not working on solving the problem during this entire period. The results of the computations for the same two students (in the same order) can be found in Figure 5. It is interesting to note that the profiles are so widely different for these students. Note that in the ones on the left there are no values for week 16. This is not because there were no loggings, but because there were no valid sequences (from T to C within the same session).

3 Experimental situation

The collection of programs that we mine were collected during the years 2003 – 2006 during a functional programming course in which `Haskell` was taught to first year students at Universiteit Utrecht. This means that we have collected programs *in vivo*, and not *in vitro*: we have had little control over the experimental situation (one might go as far as saying that there was not an experimental situation). This brings a number of complications and reservations that we shall discuss at the end of this section.

Only students who use the supplied `Helium` compiler are logged. Students can either work alone or in teams. Every compile results in a logging, and if a compilation builds imported modules, then this results in a logging for each recompiled module, and one for the module itself (which tend to be temporally quite close together of course). This applies recursively.

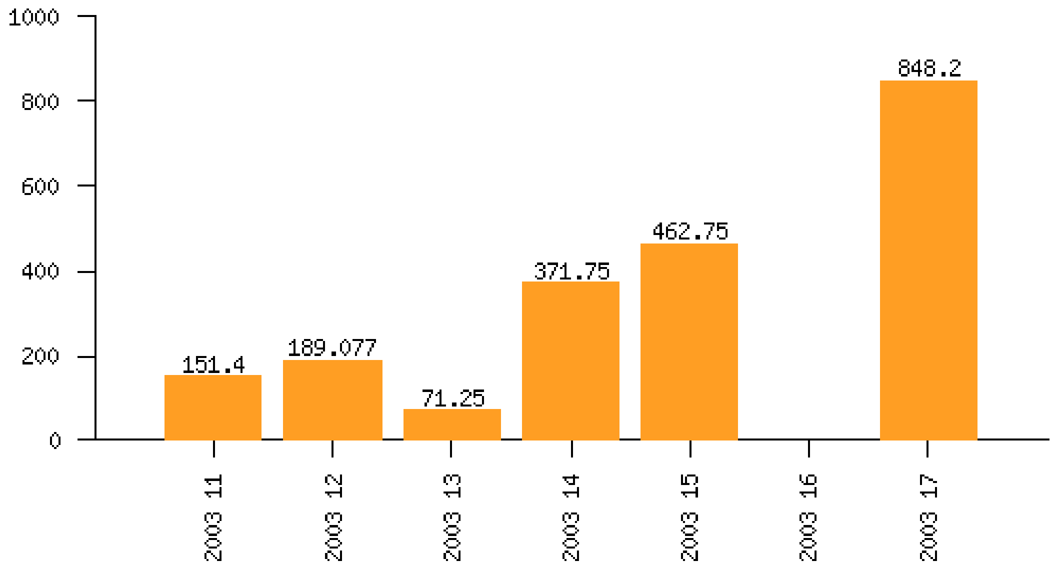
Technically, the logging is performed by communication over a socket between the compiler and a `Java` server running on the network. The `Java` server is responsible for grouping loggings originating from the same account together (this decision is based on an environment variable on the student's account,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

Over the years, we have gradually expanded what is actually logged. To begin with, we log the name of the student, the version of the compiler, and the phase in which compilation terminated (L for lexical, P for parse, S for simple static errors, T for type error, and finally C for a correct compilation). To have phase information readible available is essential, not just because the information itself is useful, but also because some analyses only apply to loggings that passed by or ended in a certain phase. For example, if we want to know how many type errors included a hint in the message (`Helium` sometimes gives explicit suggestions how a program may be corrected), then we can easily ignore the majority of programs based on the logfile alone, because it can tell us which loggings resulted in a type error.

Finally, we send the module itself, and all the modules it imports. The latter is essential to be able to recompile the module easily on the server side. As an aside, note that we do not send along modules included in the distribution of `Helium`, and we do not log the compilation of expressions entered directly in the `Helium` interpreter. `Helium` 1.6 (released early 2006) also sends along the full command-line parameters to the compiler, because some of the current parameters can influence both the phase in which compilation ends, and the error messages that the user obtains. Additionally, it also logs the `.type` files that were used during compilation.

We have obtained 29,000 loggings in 2003 from a total of 120 teams, 23,000 in 2004 from 142 teams, 11,250 in 2005 from 82 teams, and 3,800 in 2006 from 60 teams. More information about the logging facility, including its implementation, can be found in a technical report [3].

Having collected all the programs, the first issue to deal with is anonimization and clean-up.



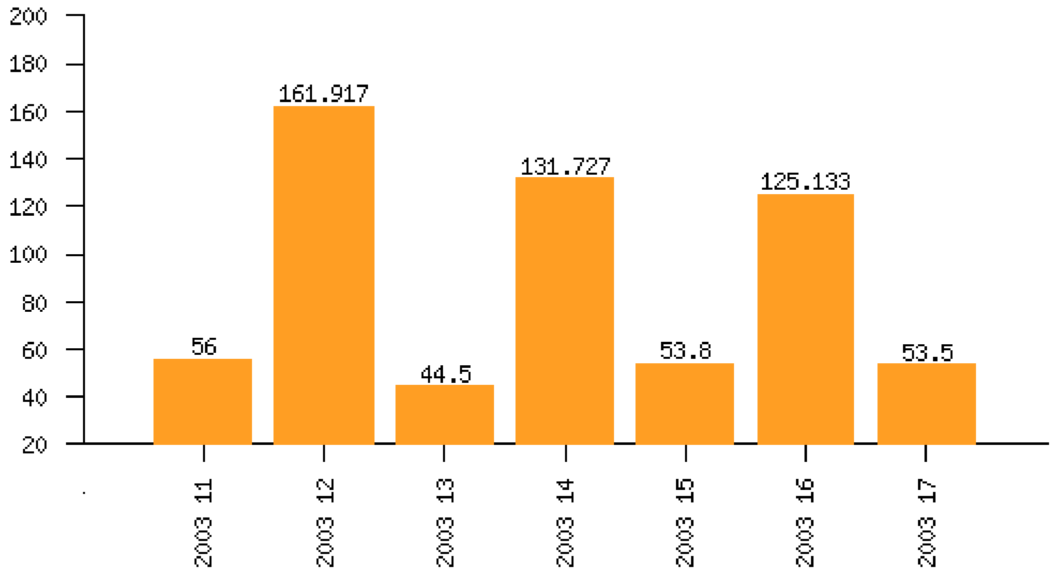


Figure 5: Average number of seconds needed to solve a type error (for two students)

Anonimization takes place by renaming the student names under which compiles were logged by a random identifier (like group0, group1,...). Because they are likely places for containing personal information, we proceed by removing all text *inside* comments and literal strings. Fortunately, hardly any analysis makes use of the contents of literal strings and comments. This process is fully automated.

To be able to reconstruct the exact error message that the students obtained, we had to reconstruct all the various versions of `Helium` that were used during the courses (eleven different compilers) and quite a bit of effort had to be made to be able to deal with differences in how the various kinds of error were represented inside the compiler, so that we can compare the kind and the number of errors made by students over the years. This is typical for the in vivo nature of our the logging process: when one starts to analyze one finds that many things have changed over the years that influence the computation of the analysis.

Especially for the earlier collections of loggings, there were some issues such as spurious compilations and missing modules (at first, we did not send the imported modules). However, we have been able to reconstruct virtually all the compiles, because in many cases, we could find the compiled module by looking a few compiles earlier. Here we could use the known phase description as a checksum, to verify that our modification was not too far off. Teams that had compiles that we could not recover were removed completely from the collection.

Although we have taken some care, the current set-up reveals a number of limitations, which implies that all our statistical results are approximate. We are confident however that given the large number of loggings we have obtained, that these can hardly influence the results. These limitation we discuss next.

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

These complications are consequences of our set-up, and we can hardly expect to get around these without changing the set-up or expanding our work. There are a number of complications that, to a large extent, can be dealt with by carefully considering the collection of loggings. Since we only log compiles done within the student network and they may do part of their work elsewhere, we have no guarantee of continuity. Indeed, since logging may be turned off, gaps in the program development may even occur even when programming is done only on the network. However, by performing structural comparisons between subsequent programs we can detect many of these cases or their absence.

Since students may work on a program in pairs, they can choose to work on different accounts during different sessions (and thus the loggings for the program will be spread over two different "teams"). We do know which students worked together (since they handed things in together) so this problem can be avoided by first merging loggings based on these names before anonimizing. Students do not always work on a single module, and it is possible that they change the name of a module during the programming process. It may even be that a student obtains a module from somewhere else, compiles and examines it, and afterwards proceeds with his own program. This kind of occurrence needs to be taken into account for some kinds of analyses (see our discussion in Section 5).

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

4 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. For example, we may first group loggings per student, and then per week to obtain a sequence of loggings for each combination of student and week.
- Computing statistical or computational characteristics of the loggings in each group. For example, we may want to compute the average number of loggings for each student over the weeks, or the maximum number of static analysis errors per compiled module over the whole collection of loggings.
- Selecting individual loggings or groups of them based on some computed characteristic.
- 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. This is essential, because otherwise a lot of time will be taken up by presenting results in a pleasing fashion. Time that could have been spent on implementing more analyses.

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. The 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 and the like.

Under these conditions, and given our familiarity with language, `Haskell` is a natural choice. Given that choice, an option would be to use a `Haskell` library for interfacing with a database management system, and this is something we shall be looking at in the future. For now, we kept the number of dependencies on other libraries as low as possible.

5 Coherence

In this section we consider a number of abstractions that we have found useful, indeed essential, for building analyses on sequences of loggings. In a sense, they can all be mapped back to concepts from Section 4 and the notion of clustering in data mining, but their importance warrants a separate description.

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

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

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

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

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

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

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

Traces can be computed by pairwise content coherence comparison of all loggings for a 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 be merged resulting in a shorter list of larger sequences. By repeatedly applying this operation to potentially all pairs of non-adjacent sequences of loggings, the set of traces for a given sequence of loggings can be obtained. Note that some care must be taken, because the end-result might depend on the order in which pairs of sequences are considered, and this also depends to some extent on the actual notion of similarity between loggings that is being used.

Finally, note that all our notions operate on sequences of loggings of some loggee, and not between loggings for different loggees. Such an extension is possible, but we do not see a need for it yet. Note though that it is perfectly possible in our uncontrolled situation that loggings that are content coherent arise from different callees, simply because the students in question work from different accounts at different times. This problem can be avoided by preprocessing as discussed in Section 3.

6 The Neon library

With a plethora of potentially interesting analyses, it is essential to offer support for doing these analyses. Therefore, we have made an implementation of a library to support building them, but also to export the resulting information to various tools for graphical presentation. In this section, we describe, at a relatively high level, the basic types and components of this library and how they can be used to compute the examples of Section 2.

For various reasons, discussed in Section 4, we have chosen to implement this library in `Haskell`. The library is built upon a small set of small generic operators and primitive analyses and ways of combining these into larger ones.

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 is then simply maps between two types of this kind:

$\textbf{type}\ \mathit{Analysis}\ \mathit{keya}\ a\ \mathit{keyb}\ b = [(\mathit{keya}, a)] \rightarrow [(\mathit{keyb}, b)]$

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 (resulting in nested lists of some kind). For example, if the values a are sequences of loggings, then after applying a grouping on logging date we obtain for b again sequences of loggings, and the key value for a given value of b shall additionally describe on which date the values belong to. Thus, the first component of each pair describes the second component. As we shall see later, this key information can also be used to generate presentations for the data automatically.

There are various ways of presenting analysis results, some via LaTeX or `HTML`, some via graphical drawing packages such as `ploticus`.

Analysis combinators

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

$$\begin{aligned}&\mathit{basicAnalysis} :: (\mathit{keya} \to \mathit{keyb}) \to\\ &\qquad\qquad\qquad (a \to b) \to\\ &\qquad\qquad\qquad \mathit{Analysis}\ \mathit{keya}\ a\ \mathit{keyb}\ b\end{aligned}$$

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.

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

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

$$\begin{aligned}&\mathit{groupAnalysis} :: (a \to \mathit{key1} \to \mathit{key2}) \to\\ &\qquad\qquad\qquad ([a] \to [[a]]) \to\\ &\qquad\qquad\qquad \mathit{Analysis}\ \mathit{key1}\ [a]\ \mathit{key2}\ [a]\end{aligned}$$

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 with the same value for 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:

$$\begin{aligned}&\mathit{selectGrpAnalysis} :: (\mathit{key1} \to \mathit{key2}) \to\\ &\qquad\qquad\qquad (a \to \mathit{Bool}) \to\\ &\qquad\qquad\qquad \mathit{Analysis}\ \mathit{key1}\ a\ \mathit{key2}\end{aligned}$$

The *selectGrpAnalysis* function selects only the groups that fulfill the predicate function, and is rather live 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 and is typically characterized by:

$$\begin{aligned}&\mathit{aggregateGrpAnalysis} :: ([a] \to [\mathit{keya}] \to \mathit{keyb}) \to\\ &\qquad\qquad\qquad ([a] \to b) \to\\ &\qquad\qquad\qquad \mathit{Analysis}\ \mathit{keya}\ a\ \mathit{keyb}\ b\end{aligned}$$

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.

Variations Although the previous types of analysis functions serve well to illustrate the basic ingredients, we do have a number slightly more general analysis functions. The added generality takes the form of a more general key transformation function. In the case of the basic analysis we add the input value as an argument to the key transformation function, in the case of the group analysis we pass the whole list of values, instead of just the first element.

```
basicAnalysis′ :: (a → keya → keyb) →
                  (a → b) →
                  Analysis keya a keyb b
groupAnalysis″ :: ([[a]] → keya → [keyb]) →
                  ([a] → [[a]]) →
                  Analysis keya [a] keyb [a]
```

This version of *groupAnalysis* is useful when defining for instance a *groupBySession* analysis, which groups a sequence of loggings into groups of time coherent sequences as described in Section 5, and adds a unique number to each session to distinguish it from the others.

```
groupPerSession =
  groupAnalysis″
    (λgrps oldkey → map (keymap oldkey) (zip [0 ..] grps))
    groupPerSession_
where
  keymap key = (λ(no, grp) →
    key ++ "; " ++ "Session: " ++ show no)
  groupPerSession_ :: [Logging] → [[Logging]]
  groupPerSession_ = ···    -- actual session grouping
```

To compose two analyses, the ◇ combinator can be used, which is conveniently similar to ordinary function composition.

```
(◇) :: Analysis keyb b keyc c →
       Analysis keya a keyb b →
       Analysis keya a keyc c
```

The ◇ combinator convienently follows the type of function composition, and is implemented in terms of it. The *mapAnalysis* combinator lifts an analysis between two types to lists of these types:

```
mapAnalysis :: Analysis keya a keyb b →
               Analysis keya [a] keyb [b]
```

The (×) operator tuples two analyses to be applied together but independently to the same input.

```
(×) :: Analysis keya a keyb1 b1 →
       Analysis keya a keyb2 b2 →
       Analysis keya a (keyb1, keyb2) (b1, b2)
```

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

```
sessionsSize = countNumberOfLoggings ◇ groupPerSession
```

To execute an analysis the function *runAnalysis* is provided. This function runs the provided analysis with the provided key and input data. Again, we only give its signature, it's implementation is straightforward:

```
runAnalysis :: a → keya →
```

```
        Analysis keya a keyb b →
        [(keyb, b)]
```

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 tables for both LaTeX and `HTML`, and to support graphical plots by generating plot files for a tool called `ploticus`. Based on the latter, we can generated graphic files that can then be included easily in documents (such as those found in Section 2).

Although this certainly speeds up the process of generating reports of our findings, we have tried to automate the process even more. Although the analysis data is often generated automatically, most tools of this kind do not support the automatic generation of textual labels for the x-axis and y-axis, figure titles, and column labels for textual presentations. We have solved this problem to some extent using a special data type for the key, called *History*. It describes exactly the operations, e.g., grouping and selecting, performed to obtain the analysis results. Based on this information, we can generate the captions needed for the figures. Indeed, the titles of Figure 3 were generated in this fashion. Because we use this so often, we have introduced special versions of our primitives to deal with this datatype. This we discuss next.

Specializations of the primitive functions

The first step towards specialization is the *Key* type class, while 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.

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

The *combineList* function is for combining a set of keys, which needs to be done when when aggregating over a set of groups. It is defined in terms of the basic *combine* function by default.

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 *Analysis* type with a new one in which the key type is fixed:

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

An example instance for keys of type *String* is:

```
instance Key String where
  startkey = ""
  combine old new = if (null old)
                    then new
                    else old ++ "; " ++ new
```

The most important abstraction, which is on top of *Key* encapsulates for each primitive function the key transformation function for the instance data type. It is defined as follows:

```
class (Key a) ⇒ DescriptiveKey a where
  basicAnakey :: String → a
  groupingKey :: (Eq b, Enum b, DataInfo b) ⇒
                 Either b (String, String) → a
  selectgrpKey :: (DataInfo b) ⇒ Maybe b → String → a
  aggregatingKey :: String → [a] → a
  tuplingkey :: a → a → a → a
  mappingkey :: String → [a] → a
```

For an instance of this type class we can, among others, use the following functions to define an analysis

```
basicAnalysis :: (DescriptiveKey key) ⇒
                 String → (a → b) → AnalysisFK key a b
groupAnalysis :: (DescriptiveKey key, Enum b, DataInfo b) ⇒
                 (a → b) → ([a] → [[a]]) →
                 AnalysisFK key [a] [a]
selectGrpAnalysis :: (DescriptiveKey key) ⇒
                     (a → Bool) →
                     AnalysisFK key a a
aggregateAnalysis :: (DescriptiveKey key) ⇒
                     String → ([a] → b) →
                     AnalysisFK key a b
(×) :: (DescriptiveKey key) ⇒
       AnalysisFK key a b1 →
       AnalysisFK key a b2 →
       AnalysisFK key a (b1, b2)
mapAnalysis :: (DescriptiveKey key) ⇒
               AnalysisFK key a b →
               AnalysisFK key [a] [b]
```

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, it may occur that one of these phases is not present in the final data. For some presentations, we should add these missing phases to result, and choose a measured value (for example, 0 in the case of integers). This is tiresome, and would be nice to automate also this aspect of generating presentations as much as possible. How this can be done for a specific type is also part of the *DataInfo* class.

An example

Examples of the above functions and the use of presentation functions utilizing the *DescriptiveKey* class instance for *KeyHistory* can be found below. Here we give some of the code for generating the right-hand side of Figure 3. We bind the analysis to *relativePhaseCountAnalysis*, mapping it over a list of logging sequences that are obtained after first grouping the input using *perStudents*. *relativePhaseCountAnalysis* itself is a straightforward composition of the steps in the analysis: first grouping per week, then apply the phase analysis to the group for each week, and afterwards The difference between using *splitAna* and *mapM* in the student grouping, and using *mapAnalysis* after the grouping per week, is that in the former case, each student gets his own bar plot. The output is a `ploticus` file to display the results as a bar plot as displayed in Figure 3. Note that the actual code in the library is a bit more complicated (due to the generation of file and directory names, and the fact that usually we do not simply generate a `ploticus` filem, but immediately embed it in an `HTML` file together with a tabular display of the information.)

```
phaseResearch :: [(KeyHistory, [Logging])] → FName → IO ()
phaseResearch input outfile = do
  let relativePhaseCountAnalysis =
        relativeValueCount
        ⋄ ordValueCount
        ⋄ (mapAnalysis phaseAnalysis)
```

```
    ◇ groupPerWeek
  barPlotRelativePhasesPerWeekPerStudent ←
    mapM ((renderBarPlot2D (AxisInfo (Right (compare))
                show) dir)
            .relativePhaseCountAnalysis)
      (splitAna $ groupPerStudent <$> input)
  writeFile outfile barPlotRelativePhasesPerWeekPerStudent
  return ()
```

Although many details are hidden in functions like *groupPerWeek*, it should be noted that these functions are generally one-liners that call one of our primitive functions, *groupAnalysis* in the case of *groupPerWeek*). The details of this are not particularly interesting, so we omit them. What is important though, is that these functions can be and are reused a lot. Indeed, it turns out that many groupings are used over and over: grouping by student, grouping by phase, grouping by session and grouping by a given time period.

7 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 investigated people programming in DITRAN, a variant of FORTRAN [9], Zelkowitz considered programs written in PL/I compiled on a mainframe at Maryland University [11], and Litecky and Davis looked at novice programmers in Cobol. We discuss these next.

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

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

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

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 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 [8] 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 [7]. 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 reported in an article by Ryder and Thompson [10]. Here they discuss the application of metrics (defined on `Haskell` programs) to investigate correlations between these metrics and bug fixes made to the program at a later stage. The application of their work is to identify positions in a program that are likely to benefit most from refactoring. Their experimental data consists of two program development histories, based on commits to a CVS repository. The main problem, as they identify themselves, is the fact that they simply do not have enough experimental data to validate their conclusions. The use of a CVS reposititory is also limiting, because usually a program that does not compile will not be commited to the repository, and most of the information we want and can obtain from our loggings has to do with how people program, and how they handle compilation errors. On the other hand, the programs they investigate actually have quite a development history, while programming assignments done by our students are done within a limited period of time, and with a fixed goal in mind.

Our main interest in their work is to 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).

It is interesting to note that there is a gap of at least 25 years between the first and the most recent studies [7] and [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. Over the following decades decentralization took place with the advent of PCs, while only in the last decade or so, is it possible to easily obtain information about compilation behaviour by adding this onto a compiler or programming environment. Note that although the studies performed by Van den Berg and his colleagues took place around 1990, and thus fall in the middle of the hiatus, their studies were done using interviews and questionnaires and not supported by tooling.

8 Discussion and future work

In this paper we have described how we have performed a statistical analysis of various properties of a large collection of logged `Haskell` programs. These can be used for various reasons: to learn about how students program and learn to program in `Haskell`, what benefits they get from using the `Helium` compiler, but also about the programming process: how much time does it take them to solve a type error? Do infinite types occur often? Are there types of errors that are particularly hard to deal with?

Since we run our analyses over a large collection of programs, one issue is how to improve efficiency and speed of the analyses. One way would be to store the programs in a database and access it using queries.

A rather simple, but very useful extension is the use of student characteristics. Although we insist on anonimity, 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? It it better to know `Prolog` instead? 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? All this information can be used in two ways: to investigate the differences implied by a given characteristic, but also to strengthen the support for a given conclusion. For example, if we know that a measure taken from the collection as a whole results in some value, and that applying it to two groups with different characteristics give approximately the same values, then we can be more confident in the validity of the measure for the whole collection. Finally, we hope that lecturers of `Haskell` around the world will consider using `Helium` and use it with logging turned on, so that we may obtain datasets collected outside our university, which will only strengthen the conclusions that may be drawn. We will be happy to help anyone interested in using `Helium` with the technical issues (hopefully providing us with the loggings!).

Although we have not considered any concrete examples yet, 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 be 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! Indeed, the first author is currently adding the logging facility to `GHC`.

Acknowledgments We thank Bastiaan Heeren and Stefan Holdermans for their support in writing this paper.

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. The Helium logging facility. Technical Report UU-CS-2005-055, Institute of Information and Computing Sciences, Utrecht University, 2005.

[4] J. Hage and B. Heeren. Heuristics for type error discovery and recovery (revised). Technical Report UU-CS-2006-007, Institute of Information and Computing Science, University Utrecht, Netherlands, July 2006. Technical Report.

[5] 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.

[6] 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.

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

[8] 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.

[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] 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.