The NEON DSEL for mining Helium programs

Jurriaan Hage

Peter van Keeken

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2007-023

www.cs.uu.nl

ISSN: 0924-3275

Abstract. Over the years we have collected a large collection of Haskell programs compiled by students with the Helium compiler. In this paper we demonstrate a DSEL called Neon written in Haskell, for computing characteristics of this collection of programs, and presenting the results visually.

1 Introduction and motivation

When the Helium compiler for learning Haskell was developed in Utrecht [3], a lot of effort was made to improve error messages. The worth of such an effort can only be established empirically, based on programs written and compiled by students. For this reason a logging facility was added to Helium. Thus far, this has resulted in about 68,000 programs, collected during various incarnations of a functional programming course.

Besides evaluating the quality of the compiler, there are at least three more reasons why one would like to query this collection of programs: To find out how students use the compiler. To discover how students learn to program (Haskell). To investigate the use of Haskell itself, e.g., are there parts of Haskell that students avoid.

Central in this paper is the Neon domain specific embedded language (DSEL), written in Haskell, that helps us deal effectively with the implementation of queries that address a particular aspect of these four categories. The main design criteria were the following: Neon should make writing queries a relatively simple and effective task. It should be based on a small set of well-understood primitives and combinators, and generate esthetically pleasing output, supporting multiple output formats, e.g., HTML tables and PNG files. We think we have succeeded in finding the right balance between these criteria. Neon is available for download [2]. It includes a small anonymized excerpt from our collected loggings for demonstration purposes.

The field we address is largely unexplored: in the early nineties researchers addressed questions similar to the ones we want to address with Neon, but the method was traditional for empirical research using, e.g., interviews [5]. More recently, Ryder and Thompson considered the use of metrics on Haskell programs to predict the location of bugs in programs [6]; their metrics are good candidates for implementation within Neon. In the Java world, BlueJ is a promising venture that analyzes the compilation behaviour of novice students [4].

The paper consists of two parts: in Section 2 we take the reader through a number of Neon queries. In Section 3 on implementation, we describe in more detail the combinators and primitives supplied by Neon. In Section 4 we discuss.

2 Case studies

We consider three case studies to illustrate how NEON can be used to data mine the collection of logged programs. In each case, we state a hypothesis to be addressed, show the query implemented with NEON to find (counter)support, and display and interpret the results of running the query on a set of loggings. The queries do not seek to address one particular hypothesis, but were selected with the sole purpose of showing what NEON offers.

Before we proceed to the analyses themselves, we first describe how analyses, analysis results and loggings are represented. An analysis result is simply 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: $[(\texttt{"st1"}, ls1), (\texttt{"st2"}, ls2)]$, 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 could very well be

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

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

NEON itself is based on a small set of primitives and combinators that can be used to implement analyses of the general polymorphic type given above. As a result, the implementation of these functions is clean and straightforward. In practice however, these general forms are hard to use, because the programmer must describe how the key and the analysis values are to be transformed. Therefore, NEON provides a special (history) key datatype KH that can faithfully describe all the primitive operations explictly, and versions of the primitives that use this datatype by default. As a result the structure of the analysis computation will be explicitly represented in the key.

Moreover, by means of a Haskell type class, $DescrK$, the changes to the key values can be described once and for all in the instance declaration

for KH. Another reason for using KH, is that it can also be used on the presentation side, for example in the automatic generation of legends and captions. Because in any transformation the key type will be the same, we typically use a simplified version of *Ana*:

type *AnaF key a b* = *Ana key a key b*

Finally then, a *Log* is a record that contains all the general information associated with a particular logging: a *username*, the *heliumVersion*, the *phase* in which the compilation ended, the *logPath* that takes us to the logged sources, and a time stamp for the logging called *logDate*.

2.1 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 is a successful compilation and code is generated (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). We hypothesize that students get more experienced in programming Haskell during the course:

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

Analysis design To find support for such a claim, the analysis calculates the number of loggings per phase, per week, and the ratio between the number of loggings per phase and the total number of loggings in each week. The ratios tell us what we want to know, while the absolute numbers are useful to get an impression of the significance of such a ratio. We present the results as stacked bar charts.

We first define *groupPerPhase* to group together loggings that ended in the same phase, using the *groupAnalysis* primitive. It uses the auxiliary *groupAllUnder* :: $Eq\ b \Rightarrow (a \to b) \to [a] \to [[a]]$ that takes a function f and list *xs* and partitions *xs* into lists, in which two values *x1* and *x2* end up in the same list if f *x1* equals f *x2*. The duplication of *phase* in both arguments to *groupAnalysis* is due to the fact that the second of these is responsible for actually grouping the arguments, while the first of these handles the automatic update of the KH value associated with a sequence of loggings.

```
groupPerPhase :: DescrK key ⇒ AnaF key [Log] [Log]
groupPerPhase = groupAnalysis phase (groupAllUnder phase)
```

The *loggingsPerPhase* analysis first selects the loggings that are associated with phases in which we are interested (as determined by *mainPhasesAnalysis*), applies *groupPerPhase* and then simply tallies the number of loggings for each phase. Note that ⋄ denotes analysis composition.

```
loggingsPerPhase ::      AnaF KH [Log] Int
loggingsPerPhase =          countNumberOfLoggings
                          ⋄ groupPerPhase
                          ⋄ mainPhasesAnalysis

mainPhasesAnalysis :: AnaF KH [Log] [Log]
mainPhasesAnalysis = basicAnalysis "" (filter (anyfrom mainphases.phase))
  where
    mainphases =  [Lexical, Parsing, Static, Typing, CodeGen]
    anyfrom    =  flip elem

countNumberOfLoggings :: DescrK key ⇒ AnaF key [a] Int
countNumberOfLoggings = basicAnalysis "number of loggings" length
```

Composing *loggingsPerPhase* with *groupPerWeek*, which groups the loggings based on the week numbers obtained from the *logDate* of each logging, yields the information we are after. The function *renderBarChartDynamic* then generates a `ploticus` file [1] which can then be used to generate Figure 1(left). (After computing the ratios, the picture on the right can be generated in a similar fashion.)

```
phaseResearch :: FilePath → [(KH, [Log])] → IO ()
phaseResearch dir input = do
  renderBarChartDynamic dir ((loggingsPerPhase ⋄ groupPerWeek) input)
  return ()
```

Interpretation Figure 1 shows that the absolute numbers vary strongly per week. For example, week 9 is in fact a holiday week, and due to small number of loggings we can decide to ignore the ratio computed for this week. The values presented in Figure 1 reveal that the number of correct compiles seem to increase up to week 10, when ignoring week 9. The increase is mostly due to a decrease in the number of parse incorrect programs. In week 11, when students start working on a new (graded) assignment, the fraction of correct compiles seems to "reset" back to week 6.

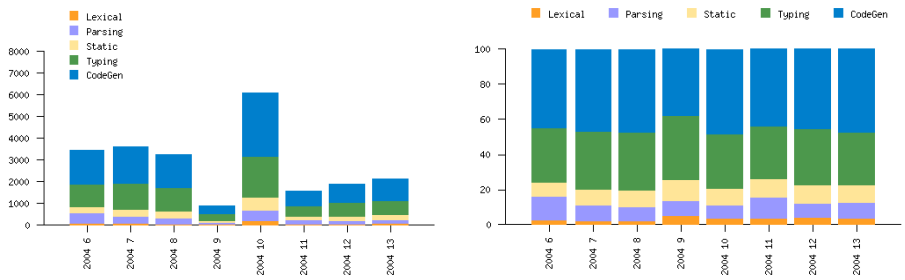

Fig. 1. Absolute and relative number of compiles per phase, given per week from 2003/2004.

2.2 A refined line count analysis

By rewarding well-documented code with a higher grade, students are stimulated to write documentation for the graded assignments as comments in the source. The idea is that writing documentation increases the understanding of the problem and the solution. However, students may decide to write the documentation at the very end. We hope for the best and state the following:

Claim: Students document their programs throughout the development process.

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

Our analysis reuses the lexer of the Helium compiler, that splits the source into a sequence of lexical tokens. A modification of this lexer analyzes the source according to the following rules: first, lines that contain both code and comments are split into separate lines, so that each split line contains only code, or only comments. For example

display str {-string to display -} = *putStrLn str* -- displayed

counts as two lines of code, one line of single line comment and one line of nested comments. Then we tally the number of code lines and comment lines, the remainder being considered empty lines. Note that due to reasons of anonymity, the contents of comments have been replaced by white space. Otherwise, the query could be refined a bit more by counting the number of non-empty comment lines.

The function *getSegments* lexes the source code and maps each line to a list of *SegmentType*s.

```
data SegmentType = CodeLn | CommentLn | EmptyLn
```

The function *lineSegmentMetric* obtains the *sourcecode* for each logging, computes the code segments, and turns this into a list of *SegmentType*s.

```
lineSegmentMetric :: AnaF KH Log [SegmentType]
lineSegmentMetric =
  basicAnalysis "Lines of source" (getSegments.sourcecode)

countSourceSegmentsMetric :: AnaF KH Log [(SegmentType, Int)]
countSourceSegmentsMetric =
    basicAnalysis "" countOrdValues
  ◇ lineSegmentMetric
```

countOrdValues :: *Ord* $a \Rightarrow [a] \rightarrow [(a, Int)]$ computes, for each value in the list, the number of times it occurs, pairs this with the value, and orders the resulting list according to the standard ordering for its type.

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

```
segmentDistribution :: ([Int] → b) → AnaF KH [Log] [(SegmentType, b)]
segmentDistribution aggregatefun =
    basicAnalysis "" (map aggregateList.(groupAllUnder fst).concat)
  ◇ mapAnalysis countSourceSegmentsMetric
where
  aggregateList list = (fst (head list), aggregatefun (map snd list))
```

The *segmentDistribution* analysis is parametrized by the aggregation function. Here we use the median, since this is considered to be a better measure to describe the central tendency of calculated values. To present the analysis results of *segmentDistribution*, we also need to compute the relative amounts. This can be done straightforwardly by *relmedianSourceLineDistribution* that applies the general *relativeCount* function to the segment distribution:

Wk	CodeLn	CommentLn	EmptyLn
6	7 (29.2%)	13 (54.2%)	4 (16.7%)
7	17 (47.2%)	9 (25.0%)	10 (27.8%)
8	40 (54.8%)	17 (23.3%)	16 (21.9%)
9	66 (62.3%)	18 (17.0%)	22 (20.8%)
10	68 (63.6%)	15 (14.0%)	24 (22.4%)
11	36 (55.4%)	14 (21.5%)	15 (23.1%)
12	42 (58.3%)	12 (16.7%)	18 (25.0%)
13	61 (55.0%)	28 (25.2%)	22 (19.8%)
14	62 (53.9%)	28 (24.3%)	25 (21.7%)

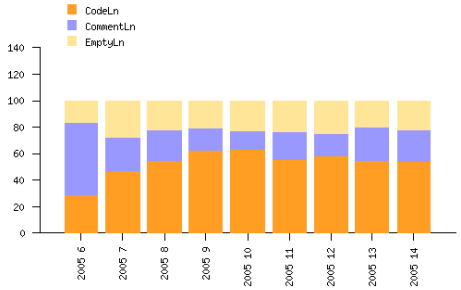

Fig. 2. Median and relative sizes of source program segments for all students, given per week from 2004/2005.

```
relmedianSourceLineDistribution =
    basicAnalysis "relative" relativeCount
  ◇ segmentDistribution median
relativeCount :: (Real b, Real c, Fractional c) ⇒ [(a, b)] → [(a, c)]
relativeCount pairs =
  let total = sum (map snd pairs)
  in  map (updateSnd (flip percentage total)) pairs
```

Interpretation The functions *renderStackedBarChart* and *showAsTable2D* can now be used to generate the table (with median and relative values) and figure (just the relative values) in Figure 2.

The median values show an increase in the total (median) program segmentation for the weeks in which an assignment was to be handed in, week 9, 10 and 14. Overall we see that comments, assumed to be documentation, is present in the logged programs in each week. In the first week, when programs are rather small, comments take more then 50% of the total source size. For the rest of the week the level of comments ranges between 15% and 25%.

2.3 Type error repair analysis

Type errors are the most occurring errors as shown by the phase analysis in Section 2.1. A more complicated analysis can be developed to investigate how long students take to repair a type incorrect program. We may expect that due to experience, this decreases over time. On the other hand, we also expect that programs become more complex, which may

result in type errors that are harder to resolve. We want to address the following claim:

Claim: The time (expressed in seconds spent) needed to solve a type error, decreases over time.

Analysis design First, we must specify how we can detect the process of correcting a type incorrect program from the logging sequence of a particular student. We take a simple approach in which we consider the first type error, and consider the sequence of loggings up to the first successful compilation, a so-called *type correcting sequences*, and we repeat. For example, denoting a lexical error, parse error, type error and successful compile with L, P, T and C respectively, we obtain $[\,T\ T\ T\ P\ T\ C\,]$ and $[\,T\ C\,]$ from $[\,C\ P\ T\ T\ T\ P\ T\ C\ L\ L\ P\ P\ T\ C\ P\,]$.

There are two important issues here that should not be forgotten: the compiles should all deal with the same program (approximated by only looking at subsequent compiles of modules with the same name), and we want to avoid overly long in between compile times: if a student makes a type error on Tuesday, and correctly compiles for the first time on Thursday, then we do not want to take that period into account. In other words, subsequent compiles in a type correcting sequences should be coherent in time and content: they are not spaced too far apart (a parameter to the function that computes the coherent sequences) and they should refer to the same module.

With this established, the remainder is quite easy: for each student, divide the complete sequence of loggings into type correcting sequences and compute the differences in time between the first and last compile in each sequence. *calcTimeAndFilenameCoherentSeqs* breaks up the logging sequence into subsequences that deal with the same source filename, in which subsequent loggings are no more than a certain number of minutes apart. It is based on the use of generally applicable *groupByCoherence* that uses two simple functions to turn a list of loggings into a list of list of loggings: *sameSourceFile* determines whether two loggings deal with the same filename, and *logTimeDiff* which determines the time between two loggings.

```
calcTimeAndFilenameCoherentSeqs :: Int → AnaF KH [Log] [[Log]]
calcTimeAndFilenameCoherentSeqs min =
  basicAnalysis ("coh.: filename and time (" ++ show min ++ ")")
                (groupByCoherence
```

```
    (λl1 l2 → sameSourceFile l1 l2
                  ∧
              logTimeDiff l1 l2 < minutesOfTD min
    ))
```

To calculate the time needed to correct a type incorrect program, we construct the *timeTCSequences* analysis. This calculates the difference in time between the first logging and the last logging of a sequence of loggings. The call to *concatMapAnalysis* is responsible for computing the type correcting sequences, as illustrated earlier. The function *timeToCompile* gathers all the ingredients together.

```
timeTCSequences :: AnaF KH [Log] TimeDiff
timeTCSequences = basicAnalysis "" (uncurry logTimeDiff.headlast)

timeToCompile :: Int → [(KH, [Log])] → [(KH, [Integer])]
timeToCompile mintime =
  ( mapAnalysis timeDiffToSecAnalysis
  ◇ mapAnalysis timeTCSequences
  ◇ concatMapAnalysis (basicAnalysis "" calculateTCSequences)
  ◇ calcTimeAndFilenameCoherentSeqs mintime
  ◇ groupPerWeek)
```

The analysis just described works on a sequence of loggings for a particular student. To apply the analysis to each student, we split up the loggings into subsequences (one for each student) and then apply the above analysis to each subsequence. This is captured generically by *analysisPerStudent* below to which we should pass *timeToCompile*.

```
analysisPerStudent :: AnaF KH [Log] a → [(KH, [Log])] → [[(KH, a)]]
analysisPerStudent ana input =
  map ana.splitAnalysis $ groupPerStudent⟨$⟩input

timeToCompilePerStudent :: Int → OutputFormat → FilePath
  → [(KH, [Log])] → IO ()
timeToCompilePerStudent mintime format outputpath input = do
  let
    name       = researchname
    researchdir = joinDirAndFileName outputpath (stringToFilePath name)
                    ++ [pathSeparator]
    analysis   = timeToCompile mintime
  createDirectoryIfMissing True researchdir
  mapM_ (renderBoxPlot researchdir) (analysisPerStudent analysis input)
```

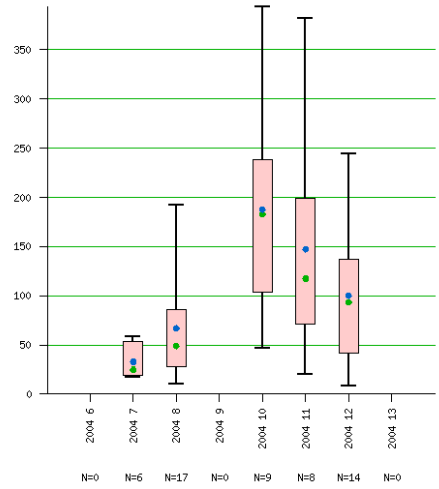

Fig. 3. Time (in seconds, 10 minutes time coherence) needed to repair a type incorrect program for a particular student, from 2003/2004.

The main difference in presenting the results of *analysisPerStudent* is that *mapM* maps the presentation function over the results.

Interpretation In Figure 3 one of the pictures generated by *timeToCompilePerStudentMain* is shown. We can see that during the second assignment period (week 10 and later) the time to solve a type error decreased. Week 6 and 9 are included in the picture, because they are part of the course, but for this particular student no type correcting sequences were found. NEON has special capabilities for dealing with such missing values (omitting them from pictures is usually not what the user wants) by means of the *DataInfo* type class. Dealing with such "missing" values can make the generation of large collections of related presentations (in this case, one for each student) an arduous task.

3 The Neon internals

This section describes the primitives and combinators, for building analyses of the kind described in Section 2. A more detailed explanation can be found in Chapters 5 and 6 of the master thesis of the second author [7]. As discussed earlier, NEON contains basic combinators that abstract away

from the particular types of key and value, and as a result their implementation is simple and clean; these are the functions that do the actual work. On top of that we have a layer of functions with similar functionality, but restricted to a single type of key, and, by means of class constraints, in some cases to particular types of key. Foremost among these is KH, that was used throughout Section 2. These functions are meant to be used by the analysis programmer.

Analysis combinators

We shall now describe some of the primitives and combinators of our library, including those that were used in Section 2. The primitives derive from the area of descriptive statistics [1], while the combinators are higher-order functions that construct analyses out of other analyses.

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:

$basicAnalysis :: (ka \to kb) \to (a \to b) \to Ana\ ka\ a\ kb\ b$
$basicAnalysis\ kf\ vf = map\ (\lambda(k, v) \to (kf\ k, kf\ v))$

The first function argument specifies how the key values should be updated, while the second actually describes the operation performed on the experimental data.

In the following 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.

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

To specify a grouping, we define the *groupAnalysis* combinator that takes a function that determines which value belong together in a group, and a function that actually performs the grouping:

$groupAnalysis :: (a \to k1 \to k2) \to ([a] \to [[a]]) \to Ana\ k1\ [a]\ k2\ [a]$

Implementation is different from *basicAnalysis* since the result of applying the value transformation 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

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

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.

Although these analysis functions serve well to illustrate the basic ingredients, we have a number of slightly more general versions, see Appendix D of [7].

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

$(\diamond) :: Ana\ kb\ b\ kc\ c \to Ana\ ka\ a\ kb\ b \to Ana\ ka\ a\ kc\ c$

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

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

Finally, we have *splitAnalysis* to run an analysis on all elements of an intermediate analysis result, as illustrated in the type error repair analysis, and *mapAnalysis* that lifts an analysis from a to b to an analysis from $[a]$ to $[b]$.

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

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

Specializations of the primitive functions

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.

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

The most important type class is *DescrK* that encapsulates a fixed (but possibly parameterized) 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 example, the *groupAnalysis* primitive now becomes:

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

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*. Note that in this case the type a is *Log* and b is the type *Phase*. The *DataInfo b* constraint is used to automatically deal with missing values: it can make sure a value for each possible phase is present in the output, even if no compile for a particular phase is present in the input. This is essential if we easily want to generate graphical pictures of this kind over a population of students: in that case we want all the pictures to show exactly the same values on the x-axis (in this case, the possible phases), and in the same order. This information is captured by the *DataInfo* type class, and is used by presentation functions such as *render1DTableSmart*.

Neon has presentation functions that assume the use of the KH datatype. Some of these generate textual output, in the form of a *MarkUpDoc* that abstracts away from a particular textual representation such as Latex or HTML (the table in Figure 2 was generated in this way).

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

Furthermore, a small number of functions are provided that turn an analysis result into a `ploticus` file, that can then be transformed into a graphical format such as PNG, PS or GIF. At this moment, bar charts, stacked bar charts, relative stacked bar charts, dynamic bar charts and box plots are supported.

$$renderBoxPlot :: (Show, Num\ b, Ord\ b) \Rightarrow \\ FilePath \to [(KH, [b])] \to IO\ (FilePath, String)$$

4 Discussion and future work

In this paper we have introduced the NEON DSEL, developed for analyzing logged Helium programs. We have shown part of what NEON offers, and have given a number of examples to show that it can be used to effectively query our collection of logged programs. NEON has the potential of yielding a large amount of information on how, e.g., students learn to program, whether hints of solving type errors actually improve programming effectiveness, or how knowledge of an imperative language helps or impedes learning Haskell.

Many of the concepts around which NEON has been built are not new. They come from the area of descriptive statistics which deals with how to summarize data. Specifically, we have made provisions for dealing with the following issues: To group loggings (repeatedly) into groups of related loggings. To compute statistical or computational characteristics of the loggings in each group. To select individual loggings or groups of them based on some computed characteristic. To present the results of our analyses in various ways.

Since many of these operations are available in database query languages such as `SQL`, the question is then why we did not use databases. Since we sometimes need to run compilers and other tools over the logged programs, it is easier to have the sources in an ordinary file system. Also, operations like comparing two programs or applying regular expressions occur often, and these are easier to express in general programming languages. A major reason for choosing Haskell is that we want easy access to Helium, in order to reuse parts of it. 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. This is apparent in all the queries we discussed. Taken together, Haskell is a natural choice. Using a Haskell library to interface with a database management system is possible, but, for now, we kept the number of dependencies on other libraries as small as possible.

One of the obvious candidates for future work, is to perform detailed studies of a particular hypothesis. Before we actually undertake such detailed analyses, a simple, but very useful extension is the use of student characteristics. For example, one would like to distinguish between students that do the course for the first time, and those that have done the course before, or to have some indication of the background of the students.

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

References

1. S. Grubb. Ploticus website. `http://ploticus.sourceforge.net`.
2. J. Hage and P. Keeken. NEON website. `http://www.cs.uu.nl/wiki/Hage/Neon`.
3. 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.
4. M. C. Jadud. A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1):25 – 40, March 2005.
5. 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.
6. 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.
7. P. van Keeken. Analyzing Helium programs obtained through logging. `http://www.cs.uu.nl/wiki/Hage/MasterStudents`.