Assessing the Quality of Evolving Haskell Systems by Measuring Structural Inequality

Sander Kamps
Open University of the Netherlands
Heerlen, The Netherlands
sanderkamps79@gmail.com

Bastiaan Heeren
Open University of the Netherlands
Heerlen, The Netherlands
bastiaan.heeren@ou.nl

Johan Jeuring
Open University of the Netherlands
Heerlen, The Netherlands
johan.jeuring@ou.nl

Abstract

Software metrics are used to measure the quality of a software system, and to understand the evolution of the system's quality over time. In this paper we report on an empirical study that investigates whether structural degradation in Haskell systems is related to decreasing software quality. For our study we use three metrics that measure internal attributes at the level of Haskell modules: intra-modular complexity (cohesion), inter-modular complexity (coupling), and module size. For these metrics, we calculate the Gini coefficient, which is a measure of the inequality in a distribution of values within a certain population, and the deviation of the population's central tendency from an empirically established ideal value. We develop a method to track the evolution, and measure the correlation between the calculated system-level information and post-release defects.

The results show that: (1) post-release defects are significantly correlated with the degree of inequality between the size of modules, (2) the inequality measure is able to indicate significant structural shifts in Haskell source code, and (3) the deviation of a population's central tendency from an ideal value can serve as a benchmark to evaluate the structural characteristics of a Haskell system. The results, however, do not show that a combined measure for inequality and ideal value deviation increases the ability to indicate the defect proneness of Haskell source code.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **General and reference** → **Empirical studies**; *Metrics*; *Measurement*.

Keywords: Software quality, Gini coefficient, Ideal value deviation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell '20, August 27, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8050-8/20/08…$15.00
https://doi.org/10.1145/3406088.3409014

ACM Reference Format:
Sander Kamps, Bastiaan Heeren, and Johan Jeuring. 2020. Assessing the Quality of Evolving Haskell Systems by Measuring Structural Inequality. In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3406088.3409014

1 Introduction

When constructing a software system, developers partition it in several smaller functional components. The organization of these components, their characteristics and their mutual interactions form the structure of the software system. The aim of software developers is to structure the software system such that it satisfies certain quality standards. However, software is not perfect. Developers writing and adapting software make errors. Every code change committed to a central repository may introduce defects, or degrade the software structure. The software structure is degrading when it progressively fails to meet the quality standards. Thus, impairing changes may decrease the quality of the software, and it is therefore important that these changes are detected so that countermeasures can be taken. By measuring the internal attributes of a software system, such as size, coupling, and cohesion, and the evolution of these attributes over time, impairing changes can be detected.

Software metrics are used to measure the internal attributes of software systems to derive external quality attributes such as modularity, maintainability, and security. The majority of these metrics measure attributes at the level of modules, classes or functions. However, effectively monitoring the quality of an entire system during its development requires information at system level. Information at system level is often obtained by aggregating the results of lower level metrics.

Popular metric aggregation methods include the mean and median. These methods provide reliable central tendency measures when the distribution of the metric values follows a Gaussian distribution. However, software metrics are seldom Gaussian distributed, but generally (highly) skewed [21, 22]. An aggregation method that provides reliable results for a skewed distribution, is the Gini coefficient [21–24]. The Gini coefficient is a method to measure the inequality in a distribution of values within a certain population. When

applied to software metrics, the inequality of the metric distribution is measured. By applying the Gini coefficient to evolving software systems, the inequality coefficient can be tracked over time, for instance to identify structural changes.

Measuring internal attributes of evolving software systems has been, and still is, the subject of much research. Through the years many metrics for measuring the structure of software systems have been proposed and used. The majority of these software systems are written in imperative or object-oriented languages. The research into the structural quality of functional languages, and in particular Haskell, is rather limited [18–20]. We are not aware of any research on the evolution of software structure with respect to software quality and, consequently, defect proneness of Haskell systems. In particular, the effect of structural inequality on the quality of Haskell systems is unknown.

To determine whether structural degradation in Haskell systems is related to decreasing software quality, we develop a method to track the evolution of these systems. The method provides system-level information about three internal attributes: intra-modular complexity (cohesion), inter-modular complexity (coupling), and module size of multiple subsequent releases. To determine the value of these attributes we have used three popular metrics that have been adapted to the Haskell language by van den Hoven [20]: these metrics are Lack of Cohesion of Methods (LCOM), Coupling Between Objects (CBO), and the Lines of Code (LOC) metrics, respectively. For each subsequent release, the distribution inequality of each internal attribute is obtained by calculating the Gini coefficient of its corresponding metric values.

In addition to the distribution inequality measure, we have developed a method that compares the system's central tendency, for each of the three metrics, to an empirically established baseline. This method thus presents system-level information that represents the system's deviation to this baseline. To define this baseline, we have analysed 135 popular Haskell systems for each of the three metrics. In particular, we want to test whether combining the inequality measure with the central tendency measure increases the strength of the correlation with the number of post-release defects. We are not aware of any research to determine this baseline for Haskell systems.

To ascertain that the system-level information accurately reflects the external quality attributes, the correlation with post-release defects of three popular Haskell systems is calculated. By observing the distribution inequality measure over multiple subsequent releases it is possible to reveal substantial changes in the system. Because change can be responsible for new defects, the detection of significant shifts in the structure of the system can be helpful as an indicator to investigate the cause of these shifts.

This paper makes the following contributions:

- We have developed an ideal value deviation function that can be used as a benchmark for Haskell systems. We elaborate on the method we use to define the function for three structural metrics in Section 4.1.
- Section 5.2 validates the relation between the distribution characteristics of intra-modular complexity (cohesion), inter-modular complexity (coupling), module size, and post-release defects.
- Section 5.3 shows that the distribution inequality measure can serve as an indicator of structural shifts in the evolution of Haskell systems.

Developers can use the two methods to monitor the effect of code changes on the software structure, and to compare the central tendency with a baseline. If a system deviates, the reason for this may be further investigated. The Gini coefficient can be used to detect whether shifts in the software structure occur, both on the short and long term. This might help in detecting the introduction of atypical modules.

The next section discusses related work on software quality during the lifetime of software systems and how this quality is measured. Section 3 describes the objectives and hypotheses of our study. Section 4 discusses the research design, and Section 5 presents the results. Section 6 covers the threats to validity. Section 7 discusses our research in relation to preceding work and suggests areas for future research. Section 8 concludes.

2 Related Work

There is a long history of research into defining software quality, finding the particular software characteristics that determine this quality, and investigating the consequences software quality has on product behaviour. Here we present an overview of work on assessing software quality, based on historical data, and on aggregating lower-level metrics to obtain system-level information.

2.1 The Relationship between Structural Quality and Error Proneness

As low quality design increases the chance that faults enter the source code, several empirical studies have assessed this relationship. Nikora and Munson [15] investigated the relationship between software structural evolution and the error count for applications written in C/C++. Their results show that structural complexity and error count increase simultaneously. This implies that structural measurements could serve as predictors of the fault proneness of the whole system. Dagpinar and Jahnke's research [7] involved different structural metrics that were used with the purpose of measuring maintainability of software systems written in Java. They report that good predictors of future maintenance activity are class size and three coupling measures. Nagappan et al. [14] looked at the post-release error history of five Microsoft software systems, written in C++ or C#, and found

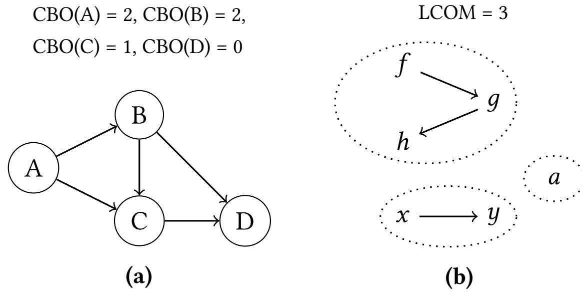


Figure 1. Module coupling and module cohesion metrics.

that fault-prone software modules are statistically correlated with software complexity measures.

2.2 Measuring Structural Quality of Haskell Systems

Van den Hoven [20] studied which metrics could be useful for determining the maintainability of Haskell programs. He defined several structural metrics for the Haskell language by adapting similar metrics from object-oriented languages. The coupling metric is based on the CBO metric, as defined by Chidamber and Kemerer [5]. The adapted CBO metric counts the number of other modules a Haskell module imports. For instance, module A imports module B and module C (see Figure 1a), and the CBO value is therefore 2. The cohesion metric is based on the LCOM4 metric, as defined by Hitz and Montazeri [11]. This fourth version of the LCOM metric counts the number of disjoint sets of functions contained within a single module. Figure 1b depicts a module with an LCOM value of 3. When a module does not contain a function the LCOM value is 0. Module size is measured by the LOC metric and is defined as the number of lines of source code, excluding comments, white space, and empty lines.

2.3 Aggregation from Lower-Level Metrics to System Level

There are several methods for aggregating metric values [13, 23, 24], each with its own advantages and drawbacks. The mean is defined as $\frac{1}{n}\sum_{i=1}^{n} x_i$, with $x_1, ..., x_n$ the multiset of metric values. The median is the value in the middle of an ordered list and is defined as $x_{\frac{n+1}{2}}$ when n is odd, and $\frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1})$ when n is even, where n is the number of ordered metric values. Both methods are simple, and give a reliable central tendency when the metric population follows a Gaussian distribution. However, the more the population deviates from the Gaussian distribution, the more unreliable these methods are.

Distribution fitting compares a metric value distribution to known distributions, such as lognormal, exponential, or power law. The parameters of known distributions are fitted to approximate the observed metric values [6]. A disadvantage of this method is that there is no widely accepted distribution available to reliably reflect software metric values [21]. Moreover, the fitting process is metric dependent, and should be repeated for every new metric [13, 23].

Software metric values and econometric values such as income inequality show similar skewed distributions. To provide more reliable and metric independent aggregation methods for software metrics, recent research has focused on using aggregation methods from econometrics.

Vasa et al. [21] first applied the Gini coefficient [9] to aggregate values from software metrics. This coefficient originates from economics and was developed for measuring income inequality. It is defined as $\frac{1}{2n^2\bar{x}}\sum_{i=1}^{n}\sum_{j=1}^{n}|x_i - x_j|$ with $x_1, ..., x_n$ the multiset of metric values and $\bar{x}$ its mean [24]. Note, however, that the Gini function is undefined for $\bar{x} = 0$. There are several reasons why the Gini coefficient can be applied to software metrics [23]. Its domain covers both positive and negative values, $\mathbb{R}^n_{x \neq 0}$, meaning that metrics with positive and negative values can also be aggregated using the Gini coefficient. The output range is bounded by [0,1] for all positive metric values, with 0 representing minimum inequality and 1 representing maximum inequality. The inverse is true for all negative metric values. This characteristic provides a way to easily compare multiple systems or system versions. For example, the Gini coefficient $G(\{50, 50, 50, 60\}) = 0.036$ and $G(\{10, 30, 100, 20\}) = 0.438$, meaning that the distribution in the first set is more equal than the second. The population size has no influence on the outcome of the Gini coefficient, which also makes it more suitable for system comparison. A main disadvantage of the Gini coefficient is that it is invariant with respect to addition and multiplication (e.g., $G(\{5, 5, 5, 6\})$ is also 0.036). As a result, it is unable to provide an indication of the absolute quality of a software system. Therefore, it is incapable of distinguishing between systems with all equally low quality components and systems with all equally high quality components. To compensate for this behaviour, the outcome should be combined with more traditional aggregation methods [23].

Bouwers et al. [3] combine the Gini coefficient with a bounded value representing the deviation from an ideal number (in their study, the ideal number of software components). They use the Gini coefficient to measure how uniformly the volume of a system is distributed over its components. The resulting value can then be used as an indicator of the system's analysability. Vasa et al. [21] have used the Gini coefficient to assess structural shifts in the distribution of class-level metric values in the evolution of several object-oriented systems. Their results show that the Gini coefficient can be used to indicate large structural shifts in software systems. Vasilescu et al. [23] applied four aggregation methods to one snapshot of a Java system to determine whether there is a correlation between class size and the prediction of defects. The result indicates a significant, but weak, correlation between three econometric aggregation methods (Gini, Theil, and Atkinson

indices) and the number of defects. They also found a significant correlation between these three methods, that is, they convey the same information. In a follow-up study [24], four aggregation methods (the three methods from their earlier work and the econometric Hoover index) were applied to two additional Java systems, showing a correlation between defects and these methods.

In our study, we also apply the Gini coefficient to aggregate module-level metrics to obtain a reliable figure for system-level information about software quality. Different from previous work, we derive the quality of software by using historical metric data and measure its correlation with post-release defects. To compensate for the invariance of the Gini coefficient under addition and multiplication, we combine it with the deviation of a system's central tendency from an ideal value. The ideal value deviation function calculates the degree to which a system's central tendency deviates from an empirically established ideal value. Our method for determining this ideal value and its upper and lower bounds is based on the approach of Bouwers et al. [3], and we elaborate on this method in Section 4.1.

3 Objectives

We want to study the evolution of software structure with respect to software quality and, consequently, defect proneness of Haskell systems. In particular, we want to study the question how structural inequality influences the quality of Haskell systems.

The Gini distribution inequality measure is combined with a more traditional aggregation method. Since the range of the Gini coefficient is bounded by [0,1], the combined measure should also lie in this range and be measured on the same scale [13]. An aggregation method that satisfies the scale and range criteria is the ideal value deviation method [3].

We define two research objectives. The first objective is to determine the values of the ideal value, lower and upper bound parameters for the ideal value deviation function. These values are determined empirically from a representative sample set of 135 Haskell systems, for each of the metrics. The second objective is to study whether structural degradation in Haskell systems is related to decreasing software quality. This is achieved by deriving system-level information from the distribution of the three metrics related to software structure, and calculating the correlation with post-release defects.

We postulate the following three hypotheses:

H1. The distribution inequality of each of the software structure related metrics (LCOM, CBO, and LOC) is correlated with the number of post-release defects.

H2. Combining distribution inequality with the ideal value deviation function increases the strength of the correlation with the number of post-release defects.

H3. The distribution inequality measure can serve as an indicator of structural shifts.

4 Empirical Research Design

The Haskell community has produced a considerable amount of source code over the years. The development and evolution of Haskell systems can be followed by looking at the commits of source code to software repositories. These commits are accompanied by meta data that provide valuable additional information such as commit dates and comments. By mining repositories, rich data sets that contain the history of software systems can be obtained.

We take the following steps to verify our hypotheses: (1) we mine the repositories of 135 open source Haskell projects (see section 4.2.1) to obtain the dataset from which we can determine the ideal value deviation function, (2) we parametrise the ideal value deviation function for the three structural metrics, (3) we mine the source code repositories of three open source Haskell projects to obtain snapshots of multiple releases; in addition, we mine their issue tracking systems for defect information, (4) we measure the correlation between the distribution inequality and post-release defects, and the distribution inequality combined with the ideal value deviation and post-release defects, and (5) we determine whether the distribution inequality measure is capable of indicating structural shifts[1].

4.1 Ideal Value Deviation Function

A key concept within software engineering is modularity. Dividing a complex system into smaller, less complex modules makes the system easier to maintain. Furthermore, each module should focus on one specific task (high cohesion) and be as independent as possible (low coupling). In general, increasing the system's modularity will adversely affect module coupling and positively affect module cohesion, and vice versa. Therefore, a trade-off must be made between the coupling between modules and the cohesion of these modules to achieve an optimal modular system [2]. Because this trade-off depends on many project-specific factors, a universal ideal value for coupling and cohesion is not available.

Also, there is no golden standard for module size. Both too large files as well as too small files have been related to the number of defects [2, 10]. Furthermore, these values may differ between programming languages [8].

As an alternative, we determine these ideal values by analysing the metric distribution of empirically obtained data from 135 Haskell repositories. Each ideal value forms a baseline to which other Haskell systems are compared. Haskell systems with a central tendency equal to the ideal value obtain the highest score of 1, while systems with a central tendency that increasingly deviates from the ideal value obtain lower scores, down to 0. Thus, besides the ideal

[1]The dataset is available at https://doi.org/10.5281/zenodo.3930143

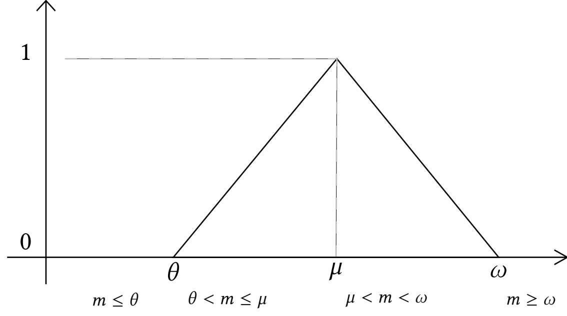


Figure 2. Plot of the ideal value deviation function.

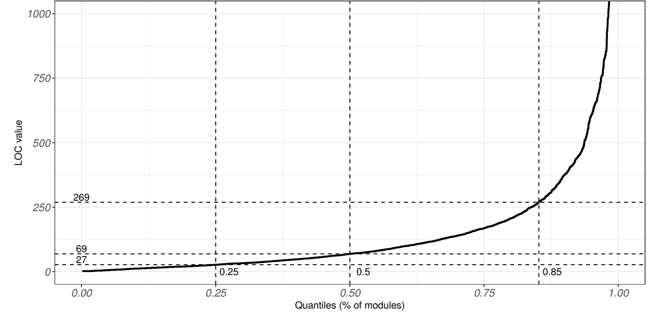


Figure 3. ECDF plot of the baseline sample set's LOC metric.

value, we have established bounds between which a system scores positive.

Similar to the method described by Bouwers et al. [3], the ideal value deviation is calculated by the function IVD ($IVD : \mathbb{R}^+ \rightarrow (0, 1)$) defined by:

$$IVD(m) = \begin{cases} \dfrac{m-\theta}{\mu-\theta} & if\ \theta < m \leq \mu \\ 1 - \dfrac{m-\mu}{\omega-\mu} & if\ \mu < m < \omega \\ 0 & if\ m \leq \theta \vee m \geq \omega \end{cases} \tag{1}$$

where parameter $\mu \in \mathbb{R}^+$ is the ideal value, $\theta \in \mathbb{R}$ is the lower bound, and $\omega \in \mathbb{R}^+$ is the upper bound. Figure 2 shows a graphical representation of the IVD function. Given an input value in the interval (θ, ω), the function returns a value in the interval $(0, 1)$. When the input value is smaller than the lower bound ($\leq \theta$) or larger than the upperbound ($\geq \omega$), the output value is 0. Thus, the output of the IVD function will gradually decrease from 1 to 0 when the input increasingly deviates from the ideal value.

4.1.1 Determining the IVD Parameters.

Since the metric distributions are not Gaussian, we cannot apply the usual methods to determine the IVD function's upper and lower bounds for these distributions. Doing so would result in a function that returns an incorrect score for systems with a central tendency deviating from the ideal value. In other words, the IVD function would not fit the shape of the Haskell system's distribution very well.

For this reason we have developed an alternative method to determine the bounds, which is based on a solid understanding of the metric distribution of the baseline sample set. The empirical cumulative distribution function (ECDF) provides a clear proportional view of a metric distribution, and is a good candidate to determine the IVD parameter values [1].

To illustrate our method, Figure 3 depicts the cumulative distribution plot of the baseline sample set's LOC metric. In this figure, the first quartile (25th percentile), the median (50th percentile), and the 85th percentile are annotated. It shows that 25% of the modules have a LOC up to 27, the median is at 69 LOC, and 15% of the modules have a LOC above 269. The IVD parameter values are then chosen such that the IVD function evaluates to a positive value (> 0), when the function's input falls within a range of values that has a considerable number of occurrences in the baseline sample set. In addition, the IVD function has discriminative power (i.e., applied to systems with central tendencies that are close together, returns distinguishable scores) and outliers are discarded.

Ideal Value (μ). We choose the median as the central tendency measure of a system. The median is resistant to outliers and is easily calculated. The median of the sample set is used to instantiate the ideal value μ.

Lower Bound (θ). For the lower bound θ, we choose a value such that the result of the IVD function is 0.5 when the first quartile of the sample set is used as the function's input. This means that a system of which the median is equal to the first quartile of the sample set receives a score of 0.5. The lower bound is then calculated by:

$$IVD(fqs) = 0.5 \implies \frac{fqs-\theta}{\mu-\theta} = 0.5 \tag{2}$$

where $fqs \in \mathbb{R}^+$ is the first quartile of the sample set. By calculating this threshold, the IVD function will evaluate to a result that is in accordance with the shape of the baseline sample set's distribution. In effect it will temper the decline of the IVD score for systems that fall within the lower range of the IVD function. Note that this can result in a negative value for the lower bound.

Upper Bound (ω). For the upper bound ω, we choose a value such that the IVD function covers a large percentage of the metric population while remaining discriminative. A trade-off must be made between these two requirements, because increased discriminative power leads to a smaller function domain. Therefore, we choose a metric value delta such that the IVD function's result is distinguishable with 1 decimal precision. For instance, a delta of 20 LOC between two systems leads to a 0.1 difference in the IVD scores. We use equation 3 to calculate the upper bound for any given

delta (Δ) and decimal precision (dp):

$$dp = \frac{\Delta}{\omega - \mu} \quad (3)$$

The upper bound for an IVD function that is discriminative to one decimal place is obtained from the following equation:

$$0.1 = \frac{\Delta}{\omega - \mu} \quad (4)$$

4.1.2 Data Acquisition. To support our empirical research, we have developed an application to analyse Haskell source code[2]. It automatically mines source code repositories and analyses the obtained Haskell source code. Next to the actual package source code, Haskell repositories often contain additional code for unit testing and Haskell files for project configuration. Because we want to focus on the core source code of Haskell systems (i.e., both end-user programs and libraries), we exclude these files from the analysis. Before analysing each package, it is purged of the occurrence of Haskell files in the project root folder and inside test folders identified by the various naming of these folders. We have checked out the master branch of each Haskell repository. The application enables us to obtain and analyse many code bases in a limited amount of time, in a consistent way.

To determine the values for θ, μ, and ω, we use a baseline sample set consisting of popular Haskell systems. Because we use the sample set to derive a benchmark against which a system will be evaluated, it is imperative that the included Haskell systems are proven to be reliable. The selected systems are publicly available and much depended upon by the Haskell community. To acquire such Haskell systems, we use the criterion that each system is a dependency for at least 100 other Haskell projects. By applying this criterion to the Haskell reverse dependency list[3], the final dataset contains the metric information of 3356 modules originating from 135 Haskell system. Due to the nature of the reverse dependency list, the set contains for a large part open source libraries.

4.2 Metric Distribution and Post-Release Defects

To ascertain whether structural degradation in Haskell systems is related to decreasing software quality, we have developed a method to track the evolution of Haskell systems. This method provides system-level information about the module size, module cohesion, and module coupling of multiple source code snapshots. Parallel to this calculation, we retrieve defect information from issue repositories.

4.2.1 Data Acquisition. We mine the repository of each system to obtain the snapshots of releases. We determine the aggregated metric values of each snapshot and map them to the number of defects submitted after a specific snapshot.

We select three systems that satisfy the following criteria:

[2]http://hackage.haskell.org/package/HaskellAnalysisProgram

[3]https://packdeps.haskellers.com/reverse | February 2019

Table 1. Dataset used in the evolution research.

System	Total nr. of snapshots	Nr. of snapshots mapped to post-release defects	Measurement period
Cabal	35	18	2012-08-14/ 2019-01-24
GHC compiler	27	21	2011-11-11/ 2019-02-05
Pandoc	124	29	2010-07-24/ 2019-04-01

1. The system is mature, which means that the system is past the (early) development stage and has had a first public release.
2. The system is actively maintained. This means that there are several releases available over a period of more than one year.
3. The complete source code revision history is available in the revision control system Git.
4. The complete issue tracking information is available.
5. The system must be non-trivial and contain at least ten parsable modules and twenty functions.

We have chosen the following three Haskell systems that satisfy these criteria: Cabal[4], the GHC compiler[5], and Pandoc[6].

To determine the number of defects reported, and subsequently map those to specific release snapshots, we acquired commit meta data and issue submission history. The snapshot meta information is available in the revision control system (Git). All three selected Haskell systems use issue tracking systems to manage issues.

To perform a comparison of post-release defects, a time frame of 60 days is applied after a release date: this time frame is long enough to collect a significant amount of issues per release, while still obtaining a large enough set of release snapshots. Issues falling outside this time frame are excluded from the measurements. Consecutive releases that fall within the time frame are also excluded from the measurements.

The final dataset contains 35 snapshots of Cabal, 27 snapshots of the GHC compiler, and 124 snapshots of Pandoc. Post-release defects are included for 18 snapshots of Cabal, 21 snapshots of the GHC compiler, and 29 snapshots of Pandoc. The content of the dataset is summarised in Table 1.

4.2.2 Combining the Gini Coefficient with the Ideal Value Deviation Function. The result of the IVD function decreases when the deviation from the ideal increases, see Figure 2. Thus, a lower result indicates a gradually less ideal metric value, and hence lower quality. This is different from the Gini coefficient, for which a lower result indicates gradually less inequality, and hence better quality. To align both methods, we use the complement of the Gini coefficient.

[4]https://github.com/haskell/cabal

[5]https://gitlab.haskell.org/ghc/ghc

[6]https://github.com/jgm/pandoc

Table 2. Six variable pairs.

Independent	Dependent
CG(LOC)	Post-release defects
CG(LCOM)	Post-release defects
CG(CBO)	Post-release defects
CG(LOC) * IVD(LOC)	Post-release defects
CG(LCOM) * IVD(LCOM)	Post-release defects
CG(CBO) * IVD(CBO)	Post-release defects

We calculate this complement by means of the function CG, defined by:

$$CG(M) = 1 - Gini(M) \quad (5)$$

The input M is the multiset of observed metric values.

We can combine CG in several ways with the ideal value deviation function (e.g. average, minimum, maximum, product, etc.). The combined value should preserve the discriminative power and accurately reflect the individual method's results. We aim for a conservative score when combining the two methods. This means that the lowest score of the two prevails over the higher score. In the extreme, for instance, CG=1 and IVD=0 will result in 0. So, the system gets the lowest possible score, although it has a perfect equality. Therefore, we choose to multiply the two values.

4.2.3 Variables.

We measure the correlation between six pairs of variables. We determine structural quality based on three metrics: LOC, LCOM, and CBO. We apply the CG function, and the CG function combined with the IVD function, to these metric values. In total, this adds up to six variables, which we consider to be the independent variables. The post-release defect count is the dependent variable. Both the independent and dependent variables are expressed on a ratio scale. The independent variable is a real number, bounded by [0,1], and the dependent variable is a whole number. Table 2 gives an overview of the variable pairs.

4.3 Structural Shifts

To determine whether the distribution inequality measure can serve as an indicator of structural shifts, we plot the progression of the distribution inequality of each metric. To confirm the measurements, we manually inspect source code to determine significant structural shifts on module level. Additionally, we have examined the release notes and commit comments to find the rationales for significant changes. We consider a shift to be significant if there is a difference of at least 4% between consecutive snapshots [21].

5 Results

In this section we present the results from the empirical research. First, we determine the IVD parameter values for the LOC, LCOM, and CBO metrics in Section 5.1. We then apply the CG and IVD functions to determine the relation between the metric distribution and post-release defects for the three selected Haskell systems (Section 5.2). Finally, we analyse significant shifts in the CG values between consecutive releases (Section 5.3).

Table 3. IVD parameter values per metric.

Metric	Lower bound θ	Ideal value μ	Upper boud ω
LOC	-15	69	269
LCOM	0	1	11
CBO	0	6	16

5.1 Parametrizing the IVD Function

Based on the metric information of 3356 modules originating from 135 Haskell systems, we have parametrised the IVD function for module volume (LOC), module cohesion (LCOM), and module coupling (CBO). The IVD parameter values for these metrics are summarised in Table 3.

5.1.1 Parameter Values for Module Volume (LOC).

Figure 4a depicts the ECDF plot of the sample set's LOC metric. The median is 69, which is used as the metric's ideal value μ. The lower bound cannot be directly obtained from the visual representation of the population, but is calculated by equation 2, $\frac{m-\theta}{\mu-\theta} = 0.5$. When for m the first quartile is used, this evaluates to $\frac{27-\theta}{69-\theta} = 0.5 \implies \theta = -15$. An acceptable trade-off for the upper bound is a delta of 20 LOC (for a 0.1 difference in the IVD score). Hence, the upper bound is calculated by equation 4: $\frac{20}{\omega-69} = 0.1 \implies \omega = 269$.

In Figure 4a we can observe that the upper bound of 269 corresponds to the 85th percentile of the sample set's LOC metric population. This means that if the system under test's median is within 85% of the sample set's observed metric values, it scores positive (> 0). We are thus able to compare systems with central tendencies that fall within this range, with 1 decimal precision. The resulting IVD function for the LOC metric is depicted graphically in Figure 4b.

5.1.2 Parameter Values for Module Cohesion (LCOM).

Figure 4c depicts the ECDF plot of the sample set's LCOM metric. Both the median and the first quartile are 1. As a consequence, calculating the lower bound with equation 2 is impossible. Therefore, we choose the lowest observed value.

Thus, the lower bound θ is set to 0. The median is 1, which we use for the ideal value μ. We choose the metric delta to be 1. Consequently, the upper bound is $\frac{1}{\omega-1} = 0.1 \implies \omega = 11$. This leads to an upper bound that is at the 94th percentile of the sample set's observed values. The resulting IVD function for module cohesion is depicted graphically in Figure 4d.

5.1.3 Parameter Values for Module Coupling (CBO).

Figure 4e depicts the ECDF plot of the sample set's CBO metric. The median and the metric's ideal value μ is 6. The lower bound is calculated by $\frac{3-\theta}{6-\theta} = 0.5 \implies \theta = 0$. The upper bound is found by $\frac{1}{\omega-6} = 0.1 \implies \omega = 16$. When selecting a delta of 1 for the upper bound, the bound is at the

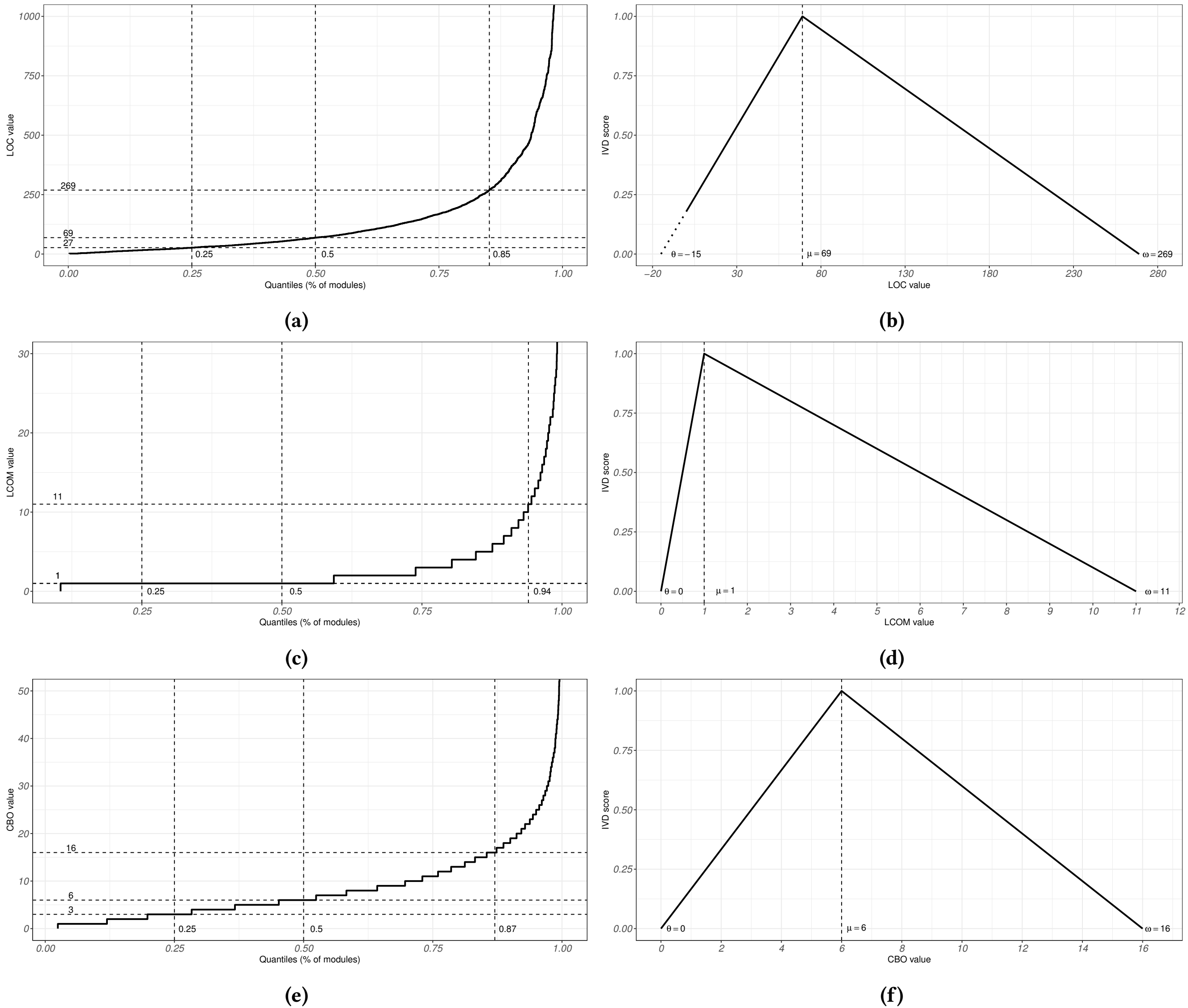


Figure 4. Cumulative distribution plot and the IVD function's graphical representation of the sample set's LOC (a and b), LCOM (c and d), and CBO (e and f) metrics.

87th percentile of the sample set's observed metric values. The resulting IVD function for module coupling is depicted graphically in Figure 4f.

5.1.4 Validation of the Chosen Parameter Values.

To validate whether the chosen parameter values result in a reasonable IVD score for each of the three metrics, we randomly pick and analyse fifteen systems from the sample system list (11% of the total number of systems). We plot the boxplot of each system's metric distribution to compare it to the distribution of the baseline sample set. We expect that systems with a similar shape and range as the sample set score close to 1. The boxplots confirm this. To illustrate our method, Figure 5 depicts the boxplots of these fifteen systems, plus the baseline sample set, for the LOC metric.

All but one of the individual systems have a more or less positively skewed distribution. The distribution of these systems thus has a similar general shape as the sample set distribution. Examples are the systems Cabal and Criterion. By contrast, system Bytestring has a deviating shape, but an interquartile range similar to the Criterion system. The median is thus further from the ideal value of 69, and as a result, Bytestring scores lower.

We chose the parameters such that the result of the IVD function, applied to a system under test, accurately reflects the degree to which that system's metric distribution deviates from the distribution of the baseline sample set.

Ten of the systems have a median less than the ideal value of 69. The IVD scores range from 0.42 for Blaze.html up to 0.94 for Criterion. Five systems have a median greater

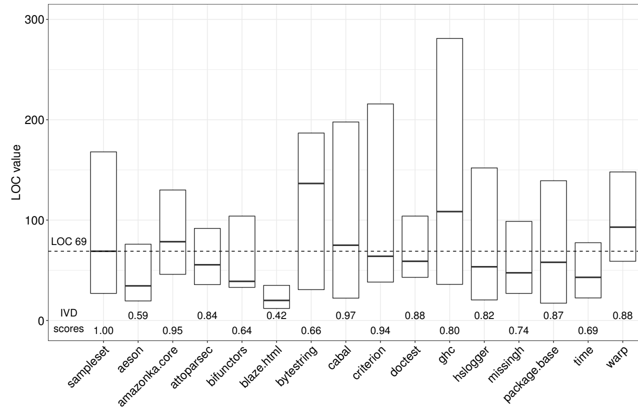


Figure 5. Boxplots of the validation set's LOC metric, showing the first quartile, median and third quartile.

than the ideal value. Their IVD scores range from 0.66 for Bytestring up to 0.97 for Cabal. We observe a more progressive decrease of the IVD scores for inputs less than the ideal value compared to inputs greater than the ideal value. However, the chosen lower bound of -15 restrains the level of decline enough to avoid disproportionately low IVD scores. On the upper half from the system median, the IVD function evaluates to a positive score (> 0) for a range of observed values that covers a substantial part of the sample sets distributions long tail, while remaining discriminative.

The discriminative power of the IVD function, in the range between the ideal value of 69 and the upper bound of 269, is 20 LOC measuring to 1 decimal place precision. For instance, Cabal and Warp have a median of 75 and 93, with an IVD score of 0.97 and 0.88, respectively.

5.2 The Relationship between Metric Distribution and Post-Release Defects

We have investigated the relationship between six pairs of variables (Table 2) for three Haskell systems. Observing the independent variables during the evolution of the three cases, we detected that there is no variance in the IVD score of the LCOM metric. It persistently stays 1. As a result of this, the CG(LCOM) and CG(LCOM) * IVD(LCOM) are identical. For this reason, we have omitted the statistical results for CG(LCOM) * IVD(LCOM).

5.2.1 Measuring the Correlation. Due to the relatively small sample sizes of the three Haskell systems, and because there is no normal distribution for any of the independent variables, we opted for using Kendall's τ correlation index for the measurement of the six variable pairs. Since we have investigated whether a structural degradation of Haskell systems is related to an increase in post-release defects (i.e., a directional relation), the correlation is tested for one-tail. The level of significance is $\alpha = 0.05$. Table 4 shows the results of the measurements of the correlation between the aggregated values and post-release defects.

5.2.2 Hypothesis 1. We accept **H1** for the LOC metric. In all three Haskell systems we observe a significant negative correlation between CG(LOC) and post-release defects. This means that as the mutual difference in size of Haskell modules increases (CG decreases), the number of reported defects also increases. We measure the strongest correlation for the GHC compiler compared to the other two. To put this in perspective, there is a 10% increase of the CG value, parallel to a decrease of approximately 40% of the number of post-release defects for the GHC compiler. For Cabal we measure the weakest correlation, 10% increase of the CG value, parallel to a decrease of approximately 20% of the number of post-release defects.

For two out of three systems (Cabal and GHC compiler), we observe a correlation between CG(LCOM) and post-release defects. Thus, in those two cases we can confirm that, when there is an increasing disparity between module's cohesion, more defects are reported.

There is no significant correlation found for the CG(CBO) variable and post-release defects for the systems Cabal and Pandoc. Only the GHC compiler system shows a significant correlation between the inequality of the module coupling distribution and post-release defects.

5.2.3 Hypothesis 2. By combining the CG function and the IVD function, the strength of the correlation between CG(CBO) * IVD(CBO) and post-release defects has increased from -.15 to -.45 for the Cabal system and from -.03 to -.62 for the Pandoc system. The variable CG(LOC) * IVD(LOC) has increased from -.34 to -.41 for the Cabal system. Although an increase is measured in these cases, we cannot accept **H2** for any of the metrics, as none of the metrics show an increase in the strength of the correlation for all three systems.

5.3 Structural Shifts

We have analysed significant shifts in the CG values between consecutive releases to test whether the distribution inequality measure can serve as an indicator of significant changes to the source code. In Figure 6 the structural shifts that exceed the 4% threshold are annotated with the delta of the CG value between consecutive releases.

We have observed that the introduction or removal of modules with characteristics that deviate from the main population clearly stand out. For instance, a significant shift of 8% and 9% for respectively the LOC and CBO values occurred at release v2.0.0.2 of the Cabal system (Figure 6a). The revision history combined with root cause analysis reveals that release v2.0.0.2 introduced many new features, improvements, and bug fixes. In this release, 166 new modules were introduced, almost doubling the number of modules. The decline for the LOC and CBO value can be explained by the

Table 4. Measurements of the correlation between the aggregated values and post-release defects.
**. Correlation is significant at the 0.01 level (1-tailed). *. Correlation is significant at the 0.05 level (1-tailed).

Cabal			GHC compiler			Pandoc		
Independent variable		Post-release defects	Independent variable		Post-release defects	Independent variable		Post-release defects
CG for LOC metric	Correlation Coefficient	-.336*	CG for LOC metric	Correlation Coefficient	-.571**	CG for LOC metric	Correlation Coefficient	-.486**
	Sig. (1-tailed)	.027		Sig. (1-tailed)	.000		Sig. (1-tailed)	.000
	N	18		N	21		N	29
CG * IVD for LOC metric	Correlation Coefficient	-.414**	CG * IVD for LOC metric	Correlation Coefficient	-.369**	CG * IVD for LOC metric	Correlation Coefficient	.373**
	Sig. (1-tailed)	.008		Sig. (1-tailed)	.010		Sig. (1-tailed)	.004
	N	18		N	21		N	29
CG for LCOM metric	Correlation Coefficient	-.336*	CG for LCOM metric	Correlation Coefficient	-.441**	CG for LCOM metric	Correlation Coefficient	.062
	Sig. (1-tailed)	.030		Sig. (1-tailed)	.003		Sig. (1-tailed)	.325
	N	18		N	21		N	29
CG for CBO metric	Correlation Coefficient	-.146	CG for CBO metric	Correlation Coefficient	-.544**	CG for CBO metric	Correlation Coefficient	-.029
	Sig. (1-tailed)	.201		Sig. (1-tailed)	.000		Sig. (1-tailed)	.417
	N	18		N	21		N	29
CG * IVD for CBO metric	Correlation Coefficient	-.452**	CG * IVD for CBO metric	Correlation Coefficient	.223	CG * IVD for CBO metric	Correlation Coefficient	-.617**
	Sig. (1-tailed)	.005		Sig. (1-tailed)	.082		Sig. (1-tailed)	.000
	N	18		N	21		N	29

introduction of smaller and less coupled modules together with an increase in size and coupling of existing modules.

At v2.2.0.0 of the Cabal system, v7.8.1 of the GHC compiler system (Figure 6b), and v1.15.1 of the Pandoc system (Figure 6c), modules with an exceptionally high LCOM value (thus, a very low cohesion) were introduced. For instance, during the development of release v7.8.1, two large modules were added to the GHC compiler with LCOM values 36 and 42. To put this in perspective, 96% of the modules had a value of less than 10. This addition caused a decline of 9%. Because the majority of the modules have a low LCOM value, the CG function is very sensitive to changes on the higher end of the scale.

The opposite, removing atypical modules from the project, is visible in the evolution of Pandoc at v1.10. In this case, two large modules were removed from the source code, causing a positive shift of 26%. At v2 they were reintroduced, causing a decrease of 6%. The impact on the degree of inequality that the reintroduction caused is less than the removal because the overall LCOM value had already increased.

We also observed improvements as a result of refactoring activities. For instance, the LCOM value at release v1.6 of Pandoc increased with 5%. This is mainly due to the refactoring of one large module, named Text.Pandoc.Shared. The release note specifically states: "Moved parsing code and ParserState from Text.Pandoc.Shared to a new module, Text.Pandoc.Parsing", thus presenting the rationale for this shift.

Another observation is the occurrences of relatively large shifts between major releases, and the relative stability in between. Major releases are recognisable by the increment of the second number, e.g., from v1.18 to v1.20. This is not entirely surprising because significant structural changes are to be expected when transitioning to the next major release. In general, the CG values remain within a compact value space. This shows that there is a bounded inequality in the metric distribution of these Haskell systems. Cabal and the GHC compiler are very similar in that respect: they both remain between 0.33 and 0.60. Pandoc shows a more volatile history, showing more frequent and larger variation in the CG values, varying between 0.40 and 0.80. Finally, we can observe a declining trend of the CG value for Cabal and the GHC compiler over their lifetime. This reveals that the metric distribution inequality grows over time.

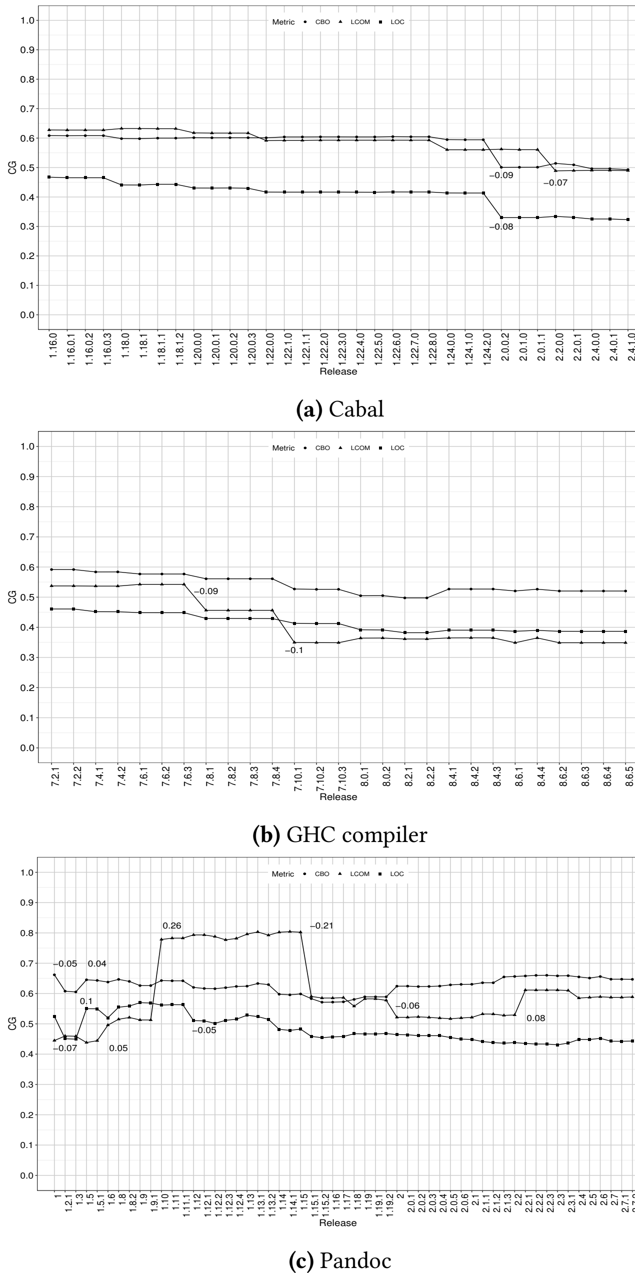
5.3.1 Hypothesis 3.

We accept **H3**: for all three Haskell systems, significant shifts in the value of the CG function are traceable to changes in the structure of the source code. By measuring the structural inequality of Haskell systems over multiple snapshots, structural changes to the source code are revealed.

6 Threats to Validity

We identify several threats to validity of this research, and describe our mitigation efforts.

Construct Validity. Selecting the appropriate metrics for measuring structural quality is paramount. Metrics for module size, module cohesion, and module coupling are used to measure the structural quality of Haskell source code. Parnas [16], Card and Agresti [4], Blundell et al. [2], and



(a) Cabal

(b) GHC compiler

(c) Pandoc

Figure 6. The distribution inequality measurement during the evolution of Cabal, the GHC compiler, and Pandoc.

Pressman [17] show that structural quality is, among others, determined by these internal attributes. Chidamber and Kemerer [5], Lorenz and Kidd [12], and Fenton and Neil [8] show that LOC, LCOM, and CBO are adequate metrics to measure these attributes. Although there are other metrics available to measure the required internal attributes, there exists a large scientific basis for the selected metrics.

Internal Validity. Our analysis software is designed and constructed to determine the value of the independent variables. Because this software is in its early experimental phase and used sparingly, scrutiny of its correct functioning is limited. Incorrect behaviour could influence the analysis results, and as a consequence, the outcome of this research.

To determine the post-release defect count, data is collected from public issue tracking systems. The use of these systems has several drawbacks. One drawback is the fickleness of issue submissions, which is hard to mitigate. Clear submission guidelines often seem to be missing and, as a consequence, submission data may provide little or incorrect information. In addition, a distinct mapping of issues to the resolving commits (e.g., an issue id) is often absent. This complicates the quantification of post-release defects as it is often unclear whether the issue actually involves a defect. Similarly, a one-to-one mapping between issue submissions and releases is often absent. Therefore, we use the release date as a reference. Issues submitted within 60 days after the release are included in the post-release defect count. We acknowledge that there is no guarantee that the submission date corresponds to the preceding release. The releases that are excluded from the measurements follow short after the included releases and are very similar in regard to the aggregated values.

Another drawback of using public issue tracking systems is that the number of discovered and reported issues may grow as a result of a growing popularity. The increase in submitted issues could then be related to the increased use, and not necessarily to the increase of implementation faults. To mitigate this effect, we have collected issues from the moment we deemed the system mature.

We have made choices for a number of variables within our research, such as lower and upper bounds and time frames. Although we have explained the rationale behind these choices, we acknowledge that other choices may lead to better results.

External Validity. To determine the relation between structural inequality and post-release defects, we analyse three popular, stable, and relatively large open source projects. Because of the limited number of subjects, and their similar characteristics, we are reluctant to claim that the results can be generalised to code bases with different characteristics.

7 Discussion and Future Work

Bouwers et al. [3] and Alves et al. [1], amongst others, have also investigated defining a benchmark, based on empirical data, with the aim to evaluate source code at program level. Although the internal quality attributes and target programming languages used in their work differ from this research, the derived benchmark thresholds are also based on the statistical properties of the underlying data. Similar to our work they developed a method that can be employed to objectively evaluate source code at system level.

We have chosen to use the system's median as input for the IVD function because of its robustness to outliers, and

because it is less affected by the skewness of the distribution than the mean. This makes it a better indicator of the central tendency of skewed distributions. However, by using the median, we only take a part of the total population into account, and ignore information about the shape of the distribution. For that reason we combine the IVD function with the CG function, which does take the whole population into account. Furthermore, the IVD function is parametrised such that it respects the positively skewed shape of the representative Haskell baseline sample set within a specific domain of metric values. Although we have shown for 15 systems that deviating distributions receive an appropriate score, this cannot be guaranteed for all cases. Further research is required to determine whether the validity of the IVD function can be improved by taking the entire population into account.

Deriving the ideal value from the baseline sample set of 135 Haskell systems has revealed that the majority of modules has a very high cohesion (median LCOM value is 1). Further research might explore the reason for this. Interesting questions would then be how Haskell relates to other programming languages, and whether this is due to the Haskell language, programming style in the community, or inherent to the functional programming paradigm.

Three Haskell projects with similar characteristics have been included in the validation of the correlation between structural quality attributes and post-release defects. The reader should bear in mind that this is a fairly limited set and, as a consequence, the generalisability of the results are limited. The analysis of more Haskell projects, with a broader range of characteristics, is required to assess our results in a more general context. We leave this as future research.

The sample sizes of the three Haskell systems' releases were fairly small. This has consequences for the power of the statistical tests we have performed. As the power decreases, the chance of not observing an effect, although it actually exists, grows (i.e., a false negative). Hence, it undermines the reliability of the results. Therefore, we are very conservative with regards to conclusions that can be distilled from these results. Because of the small sample size, we accept hypotheses 1 and 2 only if it holds for all three systems. We are less certain about the effectiveness of the metrics for which this is not the case. To increase the certainty, the sample size should be increased. Since this research is based on historical release data, from the moment these Haskell systems reached a mature state up until the present, increasing the sample size is impossible. However, as systems age, more data becomes available. This will enable future research to include larger sample sets. Another possibility would be to look at commits instead of releases, but it would be very hard to connect defects to commits.

Several studies have looked at the correlation between the change in structural quality attributes over time and post-release defects, either based on traditional aggregation methods [7, 14, 15] or based on the Gini coefficient [23, 24]. Our results confirm the conclusions of previous research and indicate that for Haskell systems the number of defects also increases directly with the increasing complexity of the source code. However, source code complexity is derived differently in each of these studies. Nikora and Munson [15] and Nagappan et al. [14] used principal component analysis to derive a set of multi-correlated metrics for object-oriented languages to measure structural quality attributes. Nagappan et al. observed that no single structural metric could be correlated with post-release defects. This finding contrasts with the results of Dagpinar and Jahnke [7], Vasilescu et al. [23, 24], and our work. In these papers, the results show a correlation with particular structural metrics and post-release defects. Hence, we hesitate to draw a general conclusion based on our findings. More research is necessary to investigate the nature of structural complexity and its relation to defect proneness of Haskell systems.

The Gini coefficient is employed to indicate structural shifts in Haskell systems and the results are similar to those obtained by Vasa et al. [21]. Our results support the idea that the Gini coefficient can be used to manage the quality of evolving software projects, since significant changes to the source code are revealed. This could, for instance, be used as a trigger to investigate the exact nature and rationale of these changes. In the study of Vasa et al. [21] and in our study, the Gini coefficient is used as the instigator to investigate changes in the source code. Whether the Gini coefficient is capable of revealing all major structural shifts still needs to be verified.

We see a declining trend of the CG values for Cabal and the GHC compiler over their lifetime and an increasing trend in the number of post-release defects. We have only measured the correlation for a set of structural characteristics and post-release defects. However, there could also be other variables that influence the increasing trend in the number of post-release defects, such as an increase in popularity. Which other variables influence the number of post-release defects remains to be determined.

8 Conclusion

This paper describes an empirical study aimed at assessing the quality of evolving Haskell systems by measuring their structural inequality. By analysing 135 representative Haskell systems, we have built a dataset, which we used to determine the ideal value, the lower bound, and the upper bound of the intra-modular complexity (cohesion), inter-modular complexity (coupling), and module size of Haskell systems. We have defined an ideal value deviation function parametrised with these results. By applying the ideal value deviation function to a sample set of 15 Haskell systems we found that the function's output was similar to the metric distribution of the 135 representative Haskell systems it was based on. This means that systems with similar distributions

receive a relatively high score, while deviating distributions receive a relatively low score. The ideal deviation function thus serves as a benchmark to evaluate the three structural quality attributes of Haskell systems.

We have used the ideal value deviation function together with a distribution inequality measure based on the Gini coefficient to investigate the quality of Haskell source code over multiple releases. Post-release defects of three Haskell systems have been collected to assess whether structural degradation is correlated with defect proneness. We found that an inequality in the metric distribution of module size is significantly correlated with post-release defects for all three of the Haskell systems studied. For two systems, we found a significant correlation with the inequality measurement of module cohesion. A correlation between post-release defects and the inequality in the metric distribution of module coupling could not be established for two of the three systems. Therefore, employing the inequality measure as an indicator of defect proneness of Haskell systems can only be verified for module size. We did not find that combining the inequality measure with the ideal value deviation function increases the strength of the correlation with post-release defects, because none of the metrics shows an increase in the strength of the correlation for all three systems. However, for two out of three systems, the strength of the correlation increased for the module coupling metric. For one system we found an increase for the module size metric.

We can use the inequality measure as an indicator of structural shifts during the evolution of Haskell systems. Significant changes to the source code have been revealed by shifts in the value of the inequality measure for all three structural metrics. Thus, the inequality measure can be employed to reveal large structural shifts and can thereby serve as an indicator of defect proneness with respect to module size.

Acknowledgments

We like to thank the anonymous reviewers for their thorough evaluation of our work and their helpful suggestions. We also thank Henrie Vos for his work on the conception and initial development of the Haskell source code analysis software we have used for our research.

References

[1] Tiago L. Alves, Christiaan Ypma, and Joost Visser. 2010. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[2] James Kenneth Blundell, Mary Lou Hines, and Jerrold Stach. 1997. The measurement of software design quality. *Annals of Software Engineering* 4 (1997), 235.

[3] Eric Bouwers, Jose Pedro Correia, Arie van Deursen, and Joost Visser. 2011. Quantifying the analyzability of software architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. IEEE, 83–92.

[4] David Noel Card and William W. Agresti. 1988. Measuring software design complexity. *Journal of Systems and Software* 8, 3 (1988), 185–197.

[5] Shyam R. Chidamber and Chris F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493.

[6] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. 2007. Power laws in a large object-oriented software system. *IEEE Transactions on Software Engineering* 33, 10 (2007), 687–708.

[7] Melis Dagpinar and Jens H. Jahnke. 2003. Predicting maintainability with object-oriented metrics an empirical comparison. In *10th Working Conference on Reverse Engineering (WCRE 2003)*. IEEE, 155–164.

[8] Norman Fenton and Martin Neil. 1999. Software metrics: Successes, failures and new directions. *Journal of Systems and Software* 47, 2 (1999), 149–157.

[9] Corrado Gini. 1921. Measurement of inequality of incomes. *The Economic Journal* 31, 121 (1921), 124–126.

[10] Les Hatton. 1997. Reexamining the fault density component size connection. *IEEE Software* 14, 2 (1997), 89–97.

[11] Martin Hitz and Behzad Montazeri. 1995. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing* (Sorrento, Italy) *(ISAAC '95)*. ISAAC, 25–27.

[12] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics : a practical guide*. Englewood Cliffs, NJ : PTR Prentice Hall.

[13] Karine Mordal, Nicolas Anquetil, Jannik Laval, Alexander Serebrenik, Bogdan Vasilescu, and Stephane Ducasse. 2013. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process* 25, 10 (2013), 1117–1135.

[14] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, 452–461.

[15] Allen P. Nikora and John C. Munson. 2003. Understanding the nature of software evolution. In *International Conference on Software Maintenance (ICSM '03)*. IEEE, 83–93.

[16] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15 (1972), 1053–1058.

[17] Roger S. Pressman. 2005. *Software engineering: a practitioner's approach*. Palgrave Macmillan.

[18] Chris Ryder and Simon J. Thompson. 2005. Software metrics measuring Haskell. In *Trends in Functional Programming*. 31–46.

[19] K. van den Berg. 1995. *Software Measurement and Functional Programming*. Ph.D. Dissertation. University of Twente.

[20] H. van den Hoven. 2015. *Invloed van structuur en samenhang op de onderhoudbaarheid van Haskell programma's (in Dutch)*. Master's thesis. Open Universiteit Nederland. http://dspace.ou.nl/handle/1820/6205

[21] Rajesh Vasa, Markus Lumpe, Philip Branch, and Oscar Nierstrasz. 2009. Comparative analysis of evolving software systems using the Gini coefficient. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 179–188.

[22] Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. 2007. The inevitable stability of software change. In *2007 IEEE International Conference on Software Maintenance (ICSM 2007)*. IEEE, 4–13.

[23] Bogdan Vasilescu, Alexander Serebrenik, and Mark van den Brand. 2010. Comparative study of software metrics' aggregation techniques. In *BENEVOL 2010 (9th Belgian-Netherlands Software Evolution Seminar, Lille, France, December 16, 2010. Proceedings of Short Papers)*. Université Lille 1, 1–5.

[24] Bogdan Vasilescu, Alexander Serebrenik, and Mark van den Brand. 2011. By No Means: A Study on Aggregating Software Metrics. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics (WETSoM '11)*. ACM, 23–26.