Measuring source code conciseness across programming languages using compression

Lodewijk Bergmans, Xander Schrijen, Edwin Ouwehand & Magiel Bruntink
Software Improvement Group, Amsterdam, The Netherlands
{l.bergmans, x.schrijen, e.ouwehand, m.bruntink}@softwareimprovementgroup.com

Abstract—It is well-known, and often a topic of heated debates, that programs in some programming languages are more concise than in others. This is a relevant factor when comparing or aggregating volume-impacted metrics on source code written in a combination of programming languages. In this paper, we present a model for measuring the conciseness of programming languages in a consistent, objective and evidence-based way. We present the approach, explain how it is founded on information theoretical principles, present detailed analysis steps and show the quantitative results of applying this model to a large benchmark of diverse commercial software applications. We demonstrate that our metric for language conciseness is strongly correlated with both an alternative analytical approach, and with a large scale developer survey, and show how its results can be applied to improve software metrics for multi-language applications.

Index Terms—benchmarked software metrics, source code volume normalization, programming language conciseness

I. INTRODUCTION

Both in the academic literature and in daily practice there are a lot of claims and opinions about the differences between programming languages. The term 'programming language wars' has even been coined to refer to the "broad divergence and impact of language designs, including often pseudo-scientific claims made that they are good or bad" [1]. One particular topic of discussion –with more opinions than evidence– is the suitability, expressiveness, or conciseness of programs in different languages (e.g. [2]–[4]).

A. Motivation

The amount of source code of systems, components or modules plays a large role in many metrics, human judging and decision making. The main reason for this is that the amount of source code in a software entity serves as a first indicator of e.g. the amount of functionality, or the effort for creating or maintaining such an entity.

In software engineering, many software metrics are used that are influenced by the amount of source code, and hence by the conciseness of that source code. For example, when identifying duplicated code, not only the amount of duplicates, but also the amount of code that is duplicated is an important factor. Or when identifying 'god classes' [5], the size of the god class is a relevant indicator for the severity of the problem. Also, at an architecture level, understanding the volume of components can be crucial for understanding the impact of issues and decisions about those components.

However, comparing the source code from different programming languages based on the number of characters or the lines of code does not do justice to both the amount of source code or the amount of work involved in that source code.

This is relevant since most modern software systems are developed using multiple programming languages. A typical example of this are different languages used for front-end development (e.g. JavaScript), business logic (e.g. Java), and data management (e.g. SQL). As a result, the application of software metrics to systems or portfolios of systems requires that the differences in conciseness between languages is taken into account.

In this paper we will investigate measures for language conciseness, not with the intention of mingling into the language expressiveness discussions, but rather with the intention of enabling software metrics that are meaningful even when they cover source code in multiple programming languages.

B. Problem statement

The problem statement for the work we present in this paper is as follows:
Given that:
- We are interested in measuring source code properties (calculating software metrics) at a system- and portfolio scale to provide insight into the quality characteristics of software.
- System- and portfolio-level metrics require the aggregation of metrics over multiple files, components and systems, written in a wide range of programming languages.
- Comparing the source code volume in different programming languages based on the number of characters or the lines of code does not do justice to both the amount of work involved or the amount of functionality in that source code.

How can we measure and compare the conciseness of source code in different languages?

C. Context and constraints

The work presented in this paper has been motivated by a practical need: at the Software Improvement Group, we run software metrics on the source code for a wide range of software applications from multiple development organizations. With currently close to 50 million lines of code analysed daily, this has to be a fully automated process (after

an intake period). Many of these applications involve multiple languages, and our analysis tools can currently handle over 250 different programming languages.

One of the key elements of this source code analysis is measuring the maintainability of software applications (according to the methodology documented in [6]–[8]), often for a portfolio of customer systems, and repeatedly over time. To properly aggregate, interpret and compare the maintainability metrics, especially in relation to the size of systems and language parts of systems, the differences in conciseness of the involved languages must be taken into account.

Hence the following constraints and considerations apply to our work:

1) The approach must be **applicable to all (text-based[1]) programming languages** we encounter. Developing and maintaining a specific semantic analysis and/or parser may not be feasible for each of these languages.
2) We aim for a **fact- or measurement-based approach**, that avoids the subjective opinions of developers (as these are known to vary wildly, and very sensitive to personal experiences and preferences). So we want to base the analysis on a source code benchmark or other data sources.
3) The approach must be **repeatable** over time, possibly with a varying set of applications, while yielding comparable results.
4) Preferably, we are **not depending on external parties** for data sources or benchmarks.

We are approaching this problem and handling these constraints from an *engineering perspective*: this means we are for example willing to trade in accuracy, when necessary to come to reasonable, repeatable and practical measures.

This paper is organized as follows: the next section presents some more background, to lay the groundwork for the rest of the paper, and discusses related work. In section III, we present the detailed steps of our approach, and in section IV, we discuss the data analysis and the metric aggregation approach. In section V, we discuss several approaches to check the validity of our results. Finally, we discuss the application of our approach to software quality metrics, and threats to validity in section VI, and summarize the contributions and some of the next steps in section VII.

II. BACKGROUND AND RELATED WORK

A. Setting the stage

To disambiguate some commonly used terms, we first provide definitions for a few core concepts: A *program* is a collection of instructions that can be executed by a computer to perform a specific task [9]. A program is specified by *source code*, following the rules of a *programming language*. In this paper we consider text-based source code only; programs may

be defined by one or more files with source code, and a single program may consist of source code in different programming languages.

A key concept behind our approach is to consider *source code as an encoding of knowledge about the program, in a format that a compiler or interpreter can process*. This knowledge spans multiple domains; from the knowledge about the programming language and its semantics, libraries that are used, coding guidelines, design decisions and design patterns, to knowledge about the requirements for the program and the application domain. It is a developer's task to gather and structure that knowledge, and encode it into source code.

Accordingly, we propose the following definitions:

program conciseness is the amount of code that is required to express a given amount of information about a program. In other words: *the ratio between the amount of information and the amount of source code for a program.*

language conciseness is the typical ratio between the amount of information and amount of code in programs written in that language. The concrete definition of 'typical ratio' will be a distinct topic in the remainder of this paper.

It should be clear that program conciseness is *not only* influenced by the programming language: other influencing factors are the application domain, the experience and background of the developers, the programming style, and coding standards, among others. The assumption is that when combining the program conciseness for a sufficient number of programs in a given language, this can be used to calculate an adequate typical conciseness indicator for that language.

Next, we will present some background on measuring information content, and then discuss related work on language conciseness.

B. On measuring information content

Above we used the term 'amount of information' to define conciseness: this is not a common metric, and may seem extremely hard to measure. However, there exists both solid theory about measuring information in a dataset (and source code is an example of such a dataset), and tested practical approaches that apply that theory.

The *Kolmogorov complexity* [10], [11] of a dataset can be described as the smallest size that a dataset can be compressed to[2]. Kolmogorov complexity is theoretically incomputable, but can be approached from above with compression algorithms [11]. For example, it has been proven that Ziv-Lempel complexity [12] or similar techniques can, assuming infinite computing power and memory, compress any object as close as possible to the theoretical minimum compressed size, without any additional knowledge about the object [13]. Indeed, there are many successful practical applications of Kolmogorov complexity that use compression algorithms. Examples are:

---

[1]Indeed, the restriction to text-based languages is a limitation, but visual and low-code programming languages have too many practical hurdles in obtaining a representative source code representation that is free from generated template structures.

[2]The more accurate definition is: the Kolmogorov complexity of an object is defined as the length of the shortest computer program (in a predetermined programming language) that produces the object as output. We will not need to use this information-theoretic definition in this paper, since we focus on the practical approximation through compression algorithms.

the comparison of (evolution of) objects [14], clustering [15] and causal inference [16]. Hence, it is safe to claim that the information content of a dataset, such as source code (which lends itself quite well to compression) can be approximated to a level that may have practical applicability.

C. Related work

Our prime motivation for measuring language conciseness is to do justice to the differences in conciseness between – source code in different– programming languages. This has been considered in its most principled form in the context of counting lines of code. In that context, the notion of *Logical Source Lines Of Code* has been coined, which aims to count statements, rather than *Physical Source Lines Of Code*; already a first abstraction from the physical representation of source code towards a more semantic measurement of volume. However, Logical Source Lines of Code does not do justice to the differences among programming languages, and how expressive statements in different languages are (or can be). Also, note that counting Logical Source Lines of Code requires parsing of the language, and hence a grammar for each language. Although the required result of this parsing is only a simple identification of statements, language features like comments and string literals must be properly handled, or the results can be completely off. Building (full) parsers can be hard and expensive, e.g. parsing C and C++ is notoriously difficult [17], as a result of macro's and preprocessing.

An important category of related (but not equivalent) work, are the language-specific function points to lines of code factors; also called *language gearing factors*. Functions points [18] is a technique that aims to estimate the size of software as a reflection of the amount of functionality a software application implements. The function points method does not require source code, but only a requirements specification, so it can already be applied at a stage where there is no software yet (although that applies more to waterfall development approaches). A related approach called Automated Function Points [19], is based on source code analysis, but still requires substantial manual input before conducting the automated process, which makes it unsuitable for large scale benchmarking.

A language gearing factor thus expresses how many statements in a given programming language are needed to express a certain amount of functionality (expressed as a function point). Hence, a language gearing factor is a measure of the expressiveness, or conciseness of the language. Language gearing factors, however, are very sensitive to reuse, both within the application, and reuse from external libraries.

Examples of sources for language gearing factors are [20]–[22]; the reported factors are based on comparing function point estimates for a benchmark of systems with the size of their source code, or by qualitative judgement of language properties [21]. Of these, the SPR table has been the most elaborate source, covering a wide range of languages. However, some of its language gearing factors are rather un-intuitive (e.g. when comparing two very similar languages with rather

different factors), and it has not been updated since 2007, hence excluding many modern (and popular) languages[3].

The notion of 'language conciseness' has a clear intuitive relation to the 'expressiveness' of programming languages. In this area, [23] is an important paper that discusses a formal framework for reasoning about the expressiveness of programming languages. This is entirely based on the formalization of language semantics, which is not applicable for the practical application we aim for in this paper. The paper does contain the following relevant statement: "By studying a number of examples we have come to the conclusion that programs in less expressive languages exhibit repeated occurrences of programming patterns and that this pattern-oriented style is detrimental to the programming process"; this does support our assumption that less expressive languages will be both less concise and require more time to program the same amount of functionality.

A final category of related work aims at identifying the conciseness or expressiveness of programming languages through empirical investigations: Berkholz [24] investigates whether the typical size of source code commits could be an indicator of programming language expressiveness, and provides a ranking of language expressiveness based on the median commit size per language for circa 50 languages. The *Hammer Principle* [25] is a large scale survey among programmers to identify a ranking of programming languages with respect to properties such as, among many others, expressiveness, verbosity and conciseness. The *Hammer Principle* survey results are also used in [26] to create rankings of various language properties. The Berkholz and *Hammer Principle* studies provide a highly relevant tool for us to validate whether our results conform to both the practices and the intuition of developers; if this is the case, it makes our results more useful (because more relatable).

In [3], an empirical investigation is conducted, where a single, small, program, was implemented in seven different languages, by 80 (student) developers. This results in comparison of the programming languages with respect to e.g. program size and programming effort. Finally, both [27] and [28] conduct a comparison of programming languages based on the *Rosetta Code* repository, which contains implementations of a set of programming tasks in a wide range of programming languages; this allows for comparing programs for the same task[4], written in multiple programming languages.

Related work on approaches to measuring language conciseness that are similar to this paper: In the domain of natural languages, research has been conducted on determining the 'linguistic complexity' of natural languages, based on the 'relative informativeness' of text samples by adopting file compression to estimate Kolmogorov complexity [13], [29].

The first version of our work was published within a Ph.D. thesis [30], where *relative* language conciseness was calculated

---

[3]At the Software Improvement Group, this has been the main driver for the reported research

[4]Although often written with different interpretations or different goals, such as brevity or readability.

based on compression ratios of source code in those languages. There it was applied to calculate the size of applications when rebuilding them in another language. Subsequently, in the M.Sc. thesis of Ouwehand [31], we investigated what the most suitable algorithms are for compressing source code of software systems, and how to build a benchmark for comparing their compression ratios.

This paper is a significant improvement over these previous works by adopting a much larger benchmark, covering more languages, presenting a novel aggregation method to calculate language conciseness, and a quantitative validation of the results with two other sources.

III. APPROACH

In this section, we will describe our benchmark-based approach to determine language conciseness, which is objective, repeatable, and applicable to any (text-based) programming language.

In section II, we already outlined the basic premises underlying our approach; the key element is to view all source code of a program, grouped per language, as data for which we can estimate the information content through data compression. By comparing the information content to the original size, a *compression ratio* is determined that is an indication of the conciseness (or verbosity) of the program. We will apply this analysis to a large benchmark of applications to aggregate to a representative *language conciseness* measure.

As the source for the benchmark, we have used an in-house software warehouse, which contains the source code of over 5000 curated industrial software applications, covering more than 280 languages (as of mid 2021), that have been scoped and analysed individually over the years.

For collecting our language conciseness benchmark, we take the following steps:

1) **Gather snapshots**: our software warehouse consists—for a large share of the systems—of many snapshots per system, over time. For each system, we use the latest version that is in the software warehouse.

2) **Select and split each system in its language parts**: to ensure that we focus on actual production source code, the files for each snapshot are scoped, which among others excludes generated code and test code. After this, the files in the system are categorized according to their language technology (or excluded if they are pure data, such as image files).

3) **Stripping comments and white-space**: to avoid the influence on compression results of natural language in source code comments, we strip all comments—and white-space—from the source code, using the `cloc` tool[5].

4) **Archiving and compression**: for each language part of a system snapshot, all source code is combined in a single tar file, which is then compressed by the LZMA2 com-pression algorithm, using the `xz` tooling[6]. An elaborate analysis of the various compression algorithms, and their ability to provide strong compression and consistency when compressing source code, has been conducted in [31]; LZMA2 was selected as the most suitable algorithm for this purpose. We use a tar archive because it allows us to compress the archived files a whole; in a zip archive, where each file is compressed separately, we would not detect redundancy across files. The latter is especially interesting since it is related to the ability of a language to achieve modularity and reuse across files. Section VI-B discusses some additional details about the choice of compression algorithms and usage of tar files.

5) **Aggregating compression results to language conciseness measures**: the previous step provides both the original size of the source code, and the compressed size: for both cases we use the number of bytes in the files; the result is a compression ratio (CR) between the original and compressed size. Next, the compression ratios for the same language from all systems are aggregated into a single language compression ratio.

We will discuss the aggregation algorithm in the next section, after showing some initial results.

IV. DATA ANALYSIS

In this section we discuss the results of the data analysis we have performed; the characteristics of the data, and some additional design decisions we made based on the findings.

Before the analysis we excluded all visual-, process- and low code-languages—i.e. languages with no text-based source code. During the analysis some systems failed to process correctly, for various implementation-related reasons. Finally, we ended up with a subset of the benchmark that had successfully passed all processing steps, consisting of 4954 systems. The size of the systems follows a log-normal distribution, with an average system size of 230 KLOC, and a median system size of 30 KLOC. The dataset contains 103 different languages[7], with an average of 3.3 languages used per system. The median number of data-points (we divide each system into distinct parts for each language; each of these parts becomes a data-point) *per language* is 24, with many data-points (over 400) for the top-ten most popular languages.

After splitting the source code for each system into its language parts (i.e. the part that collects all source code in one specific language), we obtain a benchmark with 16,314 different data-points for our analysis. Next, we take several steps of data cleaning:

1) **Remove extreme outliers**: here we remove extremely large language parts (more than 100 million characters),

---

[5]github.com/AlDanial/cloc

[6]As obtained from tukaani.org/xz/. For the results in this paper, we have used version 5.1.0alpha, with compression parameters --lzma2=preset=9e,lc=4,pb=0.

[7]Apart from the removal of visual-, process- and low code-languages, also a substantial amount of infrequently used languages did not make it to the final set, mostly because we did not include the oldest parts of our archives, or they were part of snapshots that were excluded when inconsistencies in the archive (meta-)data were found.

TABLE I: Results after each step of the data cleaning process

Step	Description	#data-points	#languages
0.	After processing	16,314	103
1.	Remove extreme outliers	15,687	102
2.	Remove CR outliers	15,017	102
3.	Remove smallest language parts	13,476	94

extremely small language parts (less than 10 lines of code), compression ratios that are less than 1, and that are over 100 (NB: note that a compression ratio *x* means there is a ratio of 1:*x* of the compressed to the original size.).

2) **Remove CR outliers**: We remove all Compression Ratio (CR) values that are more than 2 Standard Deviations from the mean. A typical cause of outlier values are (very) large duplications in the source code (e.g. identical files, folders, components or libraries), which yield high compression ratios, but are a result of design decisions (or sometimes lack thereof) by the developers rather than language characteristics.

3) **Remove smallest language parts**: We have made the design decision to remove 10% of all the smallest language parts for each language. The motivation for this is two-fold: (1) Compression algorithms naturally work less well for small datasets, if only because there is less redundancy and repeating patterns: in other words, we obtain less information about conciseness from small program parts. (2) This is especially relevant since we are interested mostly in using the results for medium-sized programs (not toy examples or small projects), and a strong influence of small language parts (which typically have a much lower compression ratio because there is inherently less redundancy in small programs). Note that a fixed size threshold for all languages would not work out well, since for some categories of languages, e.g. for shell scripts, the typical size is much smaller than for other languages, e.g. those that are often used to build enterprise systems.

Table I shows the impact on these cleaning steps on the size of the dataset, and on the number of distinct languages remaining in the dataset. Figure 1 shows box-plots of the data for a selection of languages (mostly based on how common or well-known they are): the compression ratios shown are the ratio between the compressed and the uncompressed size of the individual language parts. For each language, the number of data-points is shown in brackets. This figure shows that the CR values are distinctly different between languages: median CR values vary from circa 4 for shell scripting languages, to around 8–10 for languages such as Java and C#, to 20-25 for XML-based languages[8]. The box-plot notches, which indicate

[8]Although especially XML itself is strictly not a programming language, it is included because developers often use XML documents for configurations or specifications that are an essential part of the program behaviour: XAML, XSLT and XSD source code are good examples of such XML-based specification languages.

the 95% confidence interval for comparing the medians, show that for many languages, the median CR is distinctively different. It is expected that some languages are almost equally concise, and hence have similar medians. On the other side, for some languages the range of CRs is very similar, and although it differs quite a bit between languages, there is also a wide variation in CR values between programs, especially for the languages with a high median CR.

To provide some more insights into the compression ratios, and illustrate some further design decisions, we show the CR data for a single language, for which we have selected *Typescript*; this is a medium 'popular' language with 383 data-points in our set, after all cleaning steps.

Figure 2 shows the data-points for the *Typescript* language: for this language there were almost no outliers (marked as orange crosses), only the 10% smallest parts have been removed. The linear regression line shows that on average, larger programs are easier to compress.

One of the challenges is to determine a single CR that can be considered representative for Typescript source code. Again, one of the criteria is that the selected value is particularly representative for the mid-range sized programs. We have selected the following approach to aggregate to a single CR value:

1) First we define a linear regression line Lr over all data-points for that language, where the x-values represent code size (LOC) and the y-values the CR: this is the red dashed line in figure 2.

2) Then we select the y-value on that line at the mean code size (this corresponds to the dotted grey vertical). That y-value represents the CR for this particular language (it is marked by the solid green line).

We needed to define a threshold for the amount of data-points that is acceptable before adopting a compression ratio value. This threshold is a balance between consistency and the number of technologies for which can we produce results. After conducted a sensitivity analysis to test how many data-points are needed before obtaining a relatively stable CR value, we have observed that—at least for most languages—already after 10 data-points, the aggregated CR is very similar to the final version as derived from the full set of data-points for that language.

During the analysis of the data, we noticed that there are clear trends in the CR over time; for a majority of the languages, there was a repeated yearly incline or decline of the CR of 2-5%[9]. Given that observation, we prefer to focus on more recent data; for the final version of the compression ratios, we reduce the set of data-points by removing, per language, the data-points that are older than 5 years, *as long as* there are at least 100 data-points for that language.

Table II shows the resulting representative compression ratios for the same set of languages as in Figure 1. After all filtering steps and application of thresholds, the total list with

[9]We do not have exact explanations of the reasons for these trends, but the evolution of languages and libraries can at least partially explain such effects.
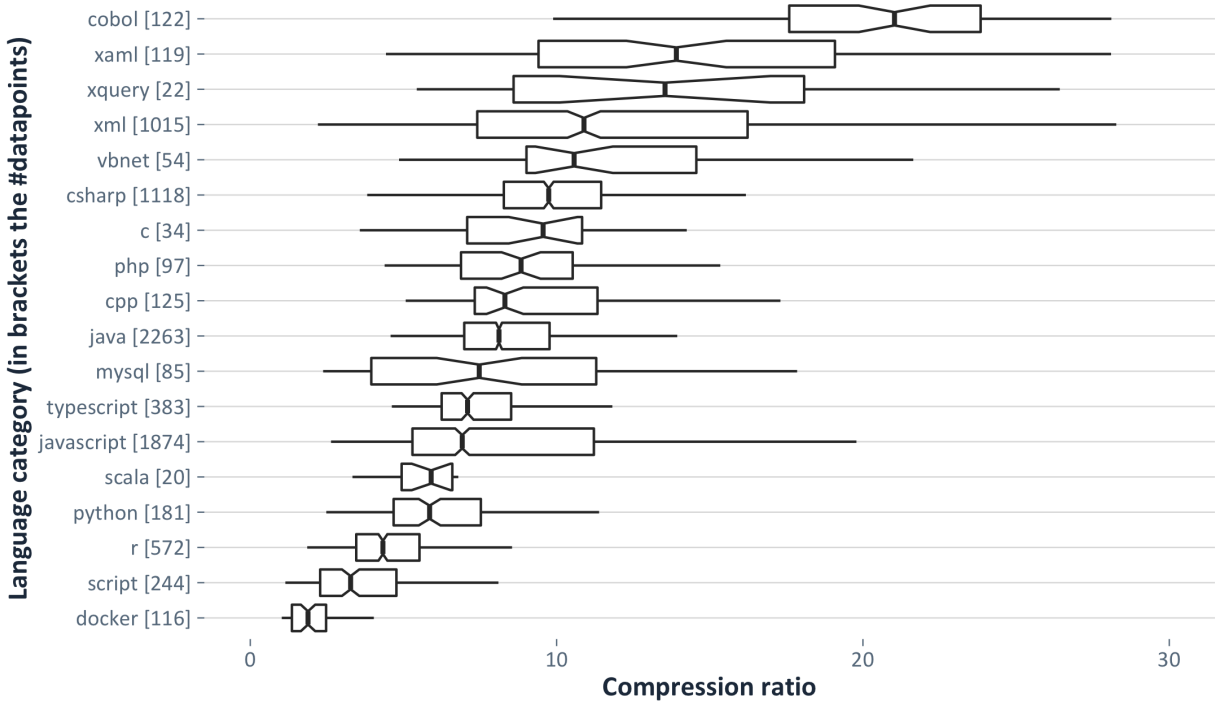
Fig. 1: Box-plot of compression ratios for 18 frequently occurring languages in our benchmark. The languages are ordered by the *median CR values*. The notches define a 95% confidence interval for comparing the medians.
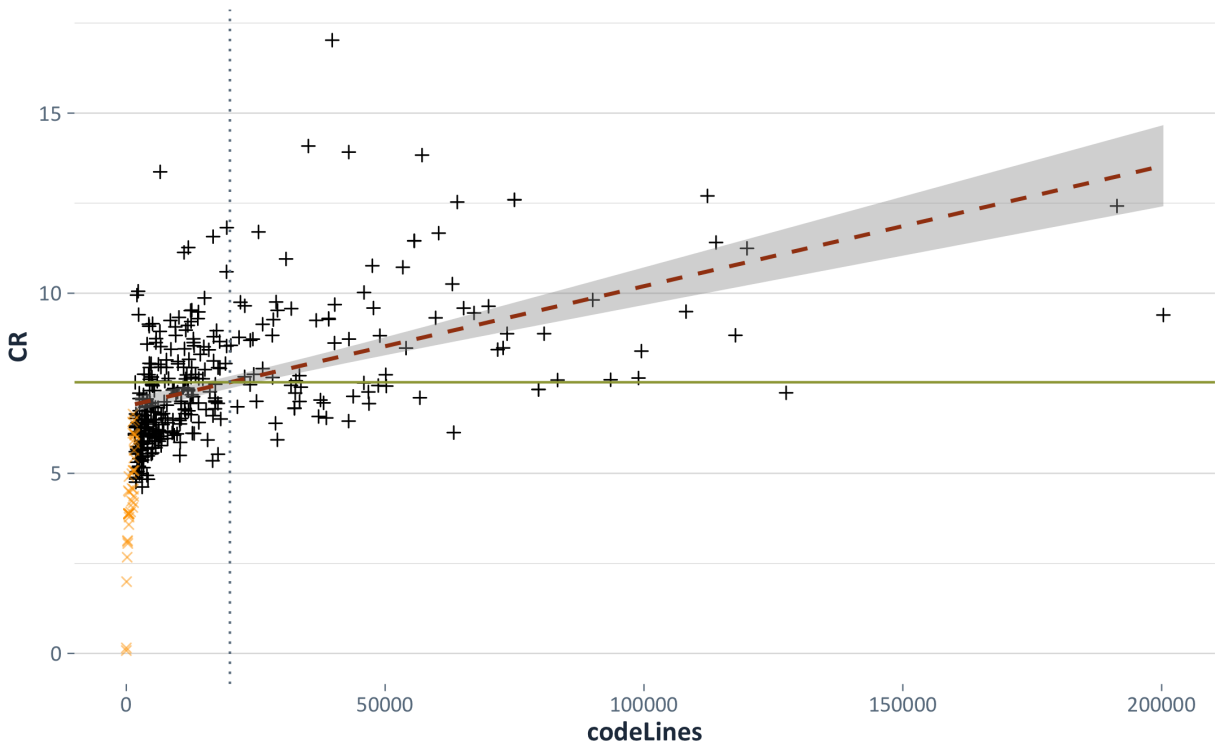

Fig. 2: A plot of all data-points for Typescript. The orange crosses are the data-points that have been removed during cleaning. The red dashed line shows a linear regression with 95% confidence interval. The grey dotted vertical line indicates the mean code-size, and the solid green line marks the calculated CR (7.5) for this language.

language compression ratios has 58 entries. CR values range from 2.2 for Docker, to 20.9 for Cobol.

TABLE II: The final Compression Ratios for a selection of languages

#	language	n	median	CR	#	language	n	median	CR
1	cobol	72	21.0	20.9	10	java	2028	8.1	8.8
2	xaml	88	13.9	14.6	11	mysql	70	7.5	8.7
3	xquery	19	13.5	13.5	12	javascript	1647	6.9	8.5
4	xml	670	10.9	12.4	13	typescript	342	7.1	7.5
5	vbnet	48	10.6	11.9	14	python	161	5.9	6.5
6	csharp	994	9.7	10.2	15	scala	17	5.9	5.6
7	cpp	111	8.3	9.8	16	r	502	4.3	4.8
8	c	28	9.6	9.3	17	script	194	3.3	4.3
9	php	82	8.8	9.3	18	docker	57	1.9	2.2

V. VALIDATION

When manually inspecting table II, at least two observations can be made:

1) The ordering of the languages (from high CRs, i.e. verbose languages, to low CRs, i.e. concise languages), does roughly follow the intuition that most developers will have: Cobol and XML variants at the top, then general purpose languages such as VB, C# and Java, followed by scripting languages such as JavaScript and Python, and functional languages such as Scala and R, followed by shell scripts at the most concise end of the spectrum.

2) Our approach results in (aggregated) CR values that are quite distinct from the median CR for that language in many cases. That might seem to imply that the aggregated CR is not very representative for the underlying data, but tests have shown that in fact it *is* quite representative: we compared our aggregation approach to several alternatives (including mean and median values); when comparing, for each data-point from our benchmark, the *Root-Mean-Squared deviation* between the calculated CR and the CR at individual data-points, our CR aggregation gave the lowest deviation, when averaging over all languages. Also the deviations had the least amount of variation across languages for our selected aggregation method. Due to lack of space, we cannot show the details of the analysis in this paper.

In section II-C, we discussed the Berkholz [24] and *Hammer Principle* [25] studies, which both provide relevant language rankings. The Berkholz ranking attempts to provide a ranking of language expressiveness; although expressiveness is not equivalent to conciseness, it is a safe assumptions that expressiveness and conciseness should correlate: in other words,

that a more expressive language is more concise (and will thus have a lower CR value). From the Berkholz dataset, we use the ranking number of the languages to identify a possible correlation with the compression ratio for those languages. Unfortunately, only 15 of the languages in the Berkholz study appear also in our final list of languages. We discuss the results below.

The *Hammer Principle* study is a survey with many[11] respondents, requesting the opinion of the respondents on many aspects of a programming language (100 questions). The survey covers 49 languages. We have selected the following six questions as relevant to the conciseness of a language:

1) "Code written in this language tends to be **verbose**."
2) "Code written in this language tends to be **terse**."
3) "This is a **low-level** language."
4) "This is a **high-level** language."
5) "This language is **expressive**."
6) "There is a lot of **accidental complexity** when writing code in this language."

Considering these six different, but related, questions also provides the opportunity to check the internal consistency in the responses. For each of these questions, all languages have been ranked from most to least 'matching' the questions (e.g. from most to least verbose). We are using the ranking number of each language for a correlation analysis with our CR values and the Berkholz ranking.

Table III shows the results of the Pearson rank correlation; generally, this analysis shows strong positive and negative correlations, that are also internally consistent (e.g. verbose and terse languages have a very strong negative correlation, and similar for high-level and low-level languages). Hence we can conclude that our CRs correlate strongly with the Berkholz language expressiveness ranking. Our CRs also have a very strong positive correlation with language verbosity, as resulting from the Hammer Principle developer survey.

From the various analyses in this section, we can conclude that:

1) The aggregated CR values are representative for the data-points in our benchmark, and show a significant distinction in conciseness for the majority of the languages.

2) The Compression Ratio of languages have a strong positive correlation (0.68) with the Berkholz programming language ranking, for the 15 languages that are overlapping both approaches.

3) According to the results from the Hammer Principle survey, the CR-based ranking of programming languages matches the perceptions of the developers quite well, especially *verbosity* and *terseness* (conciseness) are very strongly correlated (-0.84 respectively 0.82); that also matches the intention that compression ratios are a measure of conciseness. From this we conclude that, to a large extent, using CRs as an indicator of language conciseness is confirmed by (or consistent with) developer opinions.

---

[10]The number of matching languages for the correlations is 20 between compression ratio's and the Hammer Principle table, and 15 between the Berkholz ranking and the other data. The p-values for the above correlation matrix are all below 0.062; the correlations between CR and the Hammer data all have a p-value below 0.012, the correlation between CR and Berkholz has a p-value of 0.003, and the correlations between Berkholz and the Hammer data have p-values below 0.062. In other words, these correlations are quite strong and significant, even though there is only a limited set of languages overlapping between the datasets.

[11]The web page mentions "Based on responses from 398469 people, ...", although [26] mentions 13,000 in the first 2 years.

TABLE III: Spearman rank correlations between our CR values, the Berkholz ranking, and 6 different characteristics from the Hammer Principle developer survey[10].

	CR	Berkholz	Verbose	Terse	Low-level	high-level	Expressive	Acc. compl.
CR	1.00	0.68	-0.84	0.82	-0.63	0.55	0.74	-0.59
Berkholz	0.68	1.00	-0.62	0.67	-0.58	0.61	0.85	-0.49
Verbose	-0.84	-0.62	1.00	-0.95	0.69	-0.63	-0.77	0.64
Terse	0.82	0.67	-0.95	1.00	-0.65	0.57	0.86	-0.62
Low-level	-0.63	-0.58	0.69	-0.65	1.00	-0.89	-0.66	0.72
High-level	0.55	0.61	-0.63	0.57	-0.89	1.00	0.73	-0.70
Expressive	0.74	0.85	-0.77	0.86	-0.66	0.73	1.00	-0.65
Acc. compl.	-0.59	-0.49	0.64	-0.62	0.72	-0.70	-0.65	1.00

VI. DISCUSSION AND THREATS TO VALIDITY

A. Application to maintainability metrics

To illustrate how our results can be applied, we show how compression ratio's can be used for weighting of code volumes, and how this can affect the results of metrics.

To show concrete results, we calculated source code metrics for a sample system, for which we have used the open-source ANTLR implementation[12]. The main reason for choosing ANTLR is that its implementation contains substantial parts in different languages, including Java, C#, Python, JavaScript, and small amounts of Shell scripts.

Figure 3 shows in its left column the proportion of each of these languages in the system implementation, based on their respective Lines of Code. The major segment is Java, followed by C#, and two similar segments of Python and JavaScript. The segment for Scripts is so small that it does not show in the bar chart.

However, as we have argued in this paper, a line of code in one language is not comparable to a line of code in another language. In the right column of Figure 3, we show the division after weighting the LOC count with the CR values of each

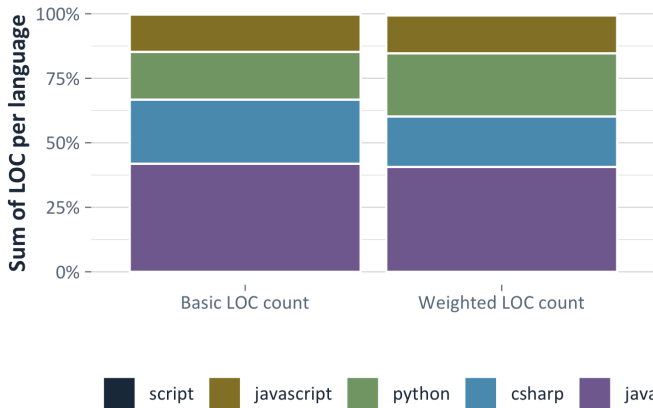[12]The source code can be found at github.com/antlr/antlr4, the version we used dates from April 19, 2019.
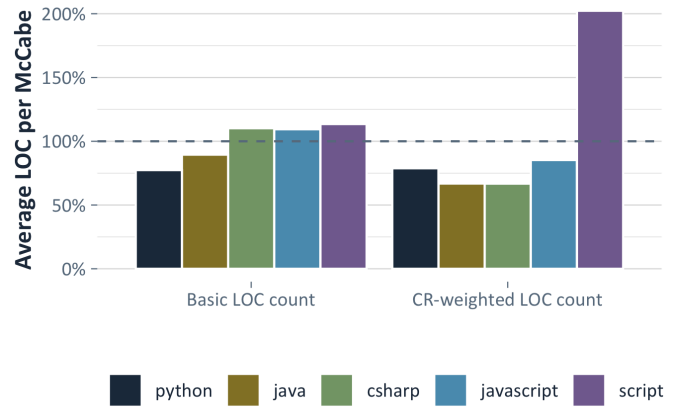

Fig. 4: Relative LOC per McCabe values for different languages, comparing the basic LOC counts on the left, with the CR-weighted LOC counts on the right.

language: weighting is done by dividing the LOC count by the compression ratio: so a line of code in a verbose language, with a high CR value, will weigh less, compared to a line of code in a more concise language, with a low CR. For the ANTLR code, this shows especially that —when applying weights— the volume of Python code is at least as large as the volume of the C# code, whereas the reverse conclusion would be drawn from the unweighted line counts. The intuitive meaning of volume based on lines of code that are weighted by CR is that it corresponds to the amount of information in that source code.

The second example, based on the same ANTLR source code, is shown in Figure 4: this shows the impact of CR-based weighting on source code metrics. As an example, we have collected the ratio between McCabe value and lines of code in functions and methods. Specifically, we collect the average of the number of lines of code per McCabe in each function, in other words, the average number of lines per decision point in the control flow. To ease comparison, Figure 4 shows these values normalized, such that 100% is the average LOC to McCabe ratio over all languages[13].

[13]This normalization makes it easier to compare the unweighted and weighted values.

Fig. 3: Proportions of language volumes in the system, based on basic, respectively weighted LOC counts.

This example shows that source code metrics that are relative to the volume of the source code (which is the case for the majority of the metrics) may lead to different conclusions when applying CR-weighting of the LOC counts. For example, in the unweighted version on the left side of Figure 4, the average LOC : McCabe ratio is very similar for C#, JavaScript and Shell scripts. But in the CR-weighted version, C# has a lower LOC : McCabe ratio than JavaScript, and Shell scripts have a ratio that is almost double, caused by the conciseness of shell scripts (and correspondingly low CR): Shell scripts have a relative low amount of decision points per line of code, when taking their conciseness into account. Also, in the unweighted version, the ratio is lower for Python than for Java, but in the weighted version this is reverse.

The examples in this subsection serve to illustrate the impact of CR-based weighting of LOC counts, and how this can lead to different insights. We have used this straightforward approach for the purpose of illustration: for the aggregation of software metrics to system level, averaging of metrics is usually not suitable, because many software metrics follow a power law distribution. We have adopted a structured approach to deal with the aggregation of software metrics such as proposed in [6] and [7], where metric values are first translated into risk categories, and then assigned a rating based on a benchmark.

Also, for presenting the weighted lines of code to developers, we do not directly use the lines of code divided by compression ratios in practice: that effectively compares source code based on the typical amount of information that it represents. However, this is not a very familiar way of thinking about the size of a program for developers. Hence, we map the volume of a program (as measured in for example lines of code) to a measure that corresponds to the amount of effort that is estimated to be required to develop that program: expressing volume as 'deduced build effort' is much easier to interpret. The actual calculation we use and its underlying design principles are beyond the scope of this paper.

B. Threats to Validity

We briefly discuss a number of (potential) threats to validity:

- *Compression algorithms are not perfect, hence they yield no perfect measure of information content*: the impact of this issue is limited because we do not use the absolute compression values per se, but use them to find a characteristic value that allows for *relative comparison* of different languages. So imperfect compression is mostly a risk if compression is systematically different between programming languages.
- *Different compression algorithms might yield different results*: Also here it is important to think of compression ratios not as absolute measurements, but as a relative comparison between programming languages. The compression ratios from different algorithms might be different, but they closely follow the same distribution. However, some compressors, such as gzip and bzip2 compress data in blocks. These are unfit for purpose

because no repetition is detected across block boundaries, which skews the results for larger systems. In [31] it has been shown that results vary by at most 1% when using compressors that do not have this mechanism.

- *Archive overhead*: A tar archive pads files with zeros to meet a certain block size. We risk skewing the result when compressing systems which are split into many smaller files. To combat this we calculate the compression ratio based on the size of the system before it is archived (so not the size of the tar archive). Because the padding compresses very well, it has no significant influence on the compressed archive size.
- *Benchmark may not be representative* Using one specific benchmark always has its limitations: in this case, the benchmark consists mostly of industrial software systems, of varying size, but including many (very) large systems. The benchmark is somewhat skewed towards bespoke administrative- and information-systems developed in North-Western Europe. We would like to point out the importance of a benchmark with real-world projects of sufficient scale; there are several repositories with algorithm implementations in multiple languages, but these are mostly unsuitable for our purposes (mostly due to the impact of small programs on compressibility).
- *Programming style may influence compressibility*: individual differences disappear by aggregating results from multiple systems from different organisations, written by many developers over the years. The scale and average size of the systems in our benchmark ensure that individual programmer differences cannot play a significant role. Note that *common* programming style differences between programming languages are, and should be, affecting the compression ratios.
- *Code structure and maintainability may influence the results*: For example, the amount of duplication in low-maintainable software. To check this, in [31], the impact of maintainability on compressibility was investigated, but no significant influence was found. The amount of modularity and reuse that languages support (either through the language features or the language ecosystems) may influence compressibility, but these are exactly the types of language characteristics that we intend to capture with our language conciseness measurements.
- *Library usage and code reuse differ significantly across languages*: This is related to the previous item, and indeed has a big impact on how much code is needed to express a certain amount of *functionality*. However, from a software maintenance perspective, it is not the amount of functionality that is the prime consideration, but the amount of information that a developer has to think about to understand, design, code and debug the software: it is the latter that our approach aims to capture.
- *Source code may contain data and natural language*: Indeed, all data constants, both numerical and text, are considered as part of the source code. However, in our benchmark analysis, we have removed lines with only

comments and white-space, to ensure the majority of the compressed data is actually source code. We assume that the amount of data constants will not differ significantly across languages, so that its effect is limited when comparing languages.

- *No ground truth to validate*: There is no objective reference to determine whether our approach is 'correct' or fully accurate. In our validation section, we have compared to both another measurement-based approach [24], and to a large developer survey [25], neither of which can be considered the truth, but both perspectives are relevant. Our motivation is mainly to have an objective approach that uses the same method to determine the conciseness factors for all programming languages.

Finally, we want to point out the risks of (wrongly) interpreting our language conciseness metrics, as e.g. shown in Table I; these language compression ratio's should not be interpreted as a rating of the 'goodness' of the languages, nor are they directly correlated to how much code is needed to implement functionality (especially libraries and frameworks influence the latter). So, dear language enthusiasts: your favourite language is not better or worse, just because it has a lower or higher compression ratio value!

VII. CONCLUSION AND FUTURE WORK

In this paper, we have investigated an evidence-based approach towards the analysis of *language conciseness*. The **first contribution** of this paper is the observation that program conciseness can be defined as the amount of code that is needed to express a single unit of information. The notion of *Kolmogorov complexity* from information theory provides a theoretical foundation for this approach *and* a practical implementation technique in the form of data compression. This technique has the big advantage that it is *fully independent of the syntax and semantics of the analysed languages*, avoiding the need for language-specific parsing of the source code. The latter is of significant importance for practical application in tooling, since the number of actively used programming languages is high, and actively growing every year.

The **second contribution** is the detailed description of the approach of data cleaning and in particular our technique for aggregating towards a language-representative value from individual data-points. We present these in sections III and IV, including many of the underlying design decisions.

The presentation of the resulting language conciseness metrics, in the form of compression ratios, for a selection of 32 languages is the **third contribution**; other researchers and practitioners may use that information for further analysis or comparison. We also share as many details of the actual inputs and results of the data analysis as space permits.

The **fourth contribution** is our validation of the results, in section V, by (a) Testing that language conciseness (CR) is representative for the systems in our benchmark, and a meaningfully distinctive metric, (b) Comparing to an evidence-based approach [24] for language expressiveness, and (c)

Confirming that our language conciseness values are highly correlated with the results of a large developer survey [25].

As we pointed out in section II-C, the fundamental ideas underlying this paper are not entirely new. But to the best of our knowledge, there has been no previous publication that documents the actual application of these ideas to a (large) benchmark covering multiple programming languages.

There are several **next steps** to follow up on the work presented here:

- We would like to extend our benchmark with an open-source version, this will provide more data-points and open up the data analysis to third parties. For the entries in such a benchmark, we may for example use [32], where a dataset of enterprise-driven open source system has been collected.
- There are further promising applications of Kolmogorov Complexity; for example the application of *Normalized Compression Distance* [33], which can provide an indication of the actual amount of code changes over time, that may be more representative than counting the changed lines of code. Normalized Compression Distance has already been applied for plagiarism detection in source code, e.g. in [34], [35].
- Finally, an interesting exercise would be to apply Normalized Compression Distance to do hierarchical clustering of programming languages (also introduced in [33]): that would yield something akin to a programming language genealogy tree[14].

Concluding, we have presented a model for determining the *language conciseness of programming languages* in a systematic and objective, data-based, way. We have presented the approach, detailed analysis steps and quantitative results of applying this model to a large representative benchmark of software applications. This has resulted in language conciseness metrics for over 50 programming languages. We have demonstrated that our metric for language conciseness is strongly correlated with both an alternative analytical approach, and a large scale developer survey.

REFERENCES

[1] A. Stefik and S. Hanenberg, "The programming language wars," in *Onward! 2014*. Portland, OR, USA: ACM, 2014.

[2] P. Graham, "Succinctness is Power," p. 93, 2002. [Online]. Available: http://www.paulgraham.com/power.html

[3] L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *IEEE Computer*, vol. 33, no. October, pp. 23–29, 2000.

[4] "SPR Programming Languages Table," Software Productivity Research, LLC, Tech. Rep. version PLT2007c, 2007.

[5] A. J. Riel, "Object-Oriented Design Heuristics," 1996.

[6] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

---

[14]See https://www.levenez.com/lang/ for an example that emphasizes the historical evolution.

[7] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability - A preliminary report," in *QUATIC 2007 - 6th International Conference on the Quality of Information and Communications Technology*, 2007, pp. 30–39.

[8] J. Visser, S. Rigal, G. Wijnholds, P. Van Eck, and R. van der Leek, *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. O'Reilly Media, Inc., 2016.

[9] Wikipedia contributors, "Computer program — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Computer_program&oldid=1013276198, 2021, [Online; accessed 1-April-2021].

[10] A. N. Kolmogorov, "On tables of random numbers," *Sankhya: The Indian Journal of Statistics, Series A*, vol. 25, pp. 369–376, 1963.

[11] M. Li and P. M. B. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications, Third Edition*, ser. Texts in Computer Science. Springer, 2008.

[12] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[13] P. Juola, "Assessing linguistic complexity," no. January 2008, pp. 89–108, 2008.

[14] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitányi, "The similarity metric," *IEEE Transactions on Information Theory*, vol. 50, no. 12, pp. 3250–3264, 2004.

[15] R. Cilibrasi, P. M. B. Vitanyi, and R. de Wolf, "Algorithmic Clustering of Music Based on String Compression," *Computer Music Journal*, vol. 28, no. 4, pp. 49–67, 2004. [Online]. Available: https://muse.jhu.edu/article/176111

[16] K. Budhathoki and J. Vreeken, "Causal inference by compression," *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 41–50, 2017.

[17] Y. Padioleau, "Parsing C/C++ Code without Pre-processing," in *Compiler Construction*, O. de Moor and M. I. Schwartzbach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 109–125.

[18] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *Software Engineering, IEEE Transactions on*, no. 6, pp. 639–648, 1983.

[19] "Automated {F}unction {P}oints ({AFP})," Object Management Group (OMG), Tech. Rep., 2014. [Online]. Available: http://www.omg.org/spec/AFP

[20] QSM, "Mendix Benchmark Report," no. September, pp. 1–30, 2014. [Online]. Available: https://www.gillz.nl/wp-content/uploads/2017/03/QSM-Mendix-Benchmark-Report.pdf

[21] C. Jones, "Programming Languages Table, Version PLT2007c," *Software Productivity Research, Burlington, MA*, 2007.

[22] ——, "The Mess of Software Metrics," Namcook Analytics, Tech. Rep., 2017. [Online]. Available: https://namcook.com/articles/TheMessofSoftwareMetrics2017.pdf

[23] M. Felleisen, "On the Expressive Power of Programming Languages," *Science of Computer Programming*, vol. 17, no. 1-3, pp. 35–75, 1991.

[24] D. Berkholz, "Programming languages ranked by expressiveness," 2013. [Online]. Available: https://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/

[25] D. R. MacIver, "Hammer principle: expressiveness of Programming languages," 2017. [Online]. Available: https://web.archive.org/web/20171012104831/http://www.hammerprinciple.com/therighttool

[26] L. A. Meyerovich and A. Rabkin, "How not to survey developers and repositories: Experiences analyzing language adoption," *SPLASH 2012: PLATEAU 2012 - Proceedings of the 2012 ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 7–16, 2012.

[27] (Wolfram), "How the Wolfram Language Measures Up," 2014. [Online]. Available: https://blog.wolfram.com/2014/06/04/how-the-wolfram-language-measures-up/

[28] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," *Proceedings - International Conference on Software Engineering*, vol. 1, pp. 778–788, 2015.

[29] K. Ehret and B. Szmrecsanyi, "An information-theoretic approach to assess linguistic complexity," *Complexity, Isolation, and Variation*, pp. 71–94, 2016.

[30] S. Raemaekers, "Origin, Impact and Cost of Interface Instability," PhD thesis, University of Delft, The Netherlands, 2015. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid:dbb70852-e06b-40f7-b872-60047f962dbc?collection=research

[31] E. Ouwehand, "Language-independent volume measurement," Master's thesis, University of Amsterdam, The Netherlands, 2018.

[32] D. Spinellis, Z. Kotti, K. Kravvaritis, G. Theodorou, and P. Louridas, "A Dataset of Enterprise-Driven Open Source Software," 2020. [Online]. Available: http://arxiv.org/abs/2002.03927

[33] R. Cilibrasi and P. M. B. Vitanyi, "Clustering by compression," *Information Theory, IEEE Transactions on*, vol. 51, no. 4, pp. 1523–1545, apr 2005.

[34] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared information and program plagiarism detection," *IEEE Transactions on Information Theory*, vol. 50, no. 7, pp. 1545–1551, 2004.

[35] I. Pribela, G. Rakić, and Z. Budimac, "Detecting source code similarity using compression," *CEUR Workshop Proceedings*, vol. 2508, no. September, pp. 22–25, 2019.