A Comparative Study of Programming Languages in Rosetta Code

Sebastian Nanz · Carlo A. Furia

Chair of Software Engineering, Department of Computer Science, ETH Zurich, Switzerland

firstname.lastname@inf.ethz.ch

Abstract—Sometimes debates on programming languages are more religious than scientific. Questions about which language is more succinct or efficient, or makes developers more productive are discussed with fervor, and their answers are too often based on anecdotes and unsubstantiated beliefs. In this study, we use the largely untapped research potential of Rosetta Code, a code repository of solutions to common programming tasks in various languages, which offers a large data set for analysis. Our study is based on 7'087 solution programs corresponding to 745 tasks in 8 widely used languages representing the major programming paradigms (procedural: C and Go; object-oriented: C# and Java; functional: F# and Haskell; scripting: Python and Ruby). Our statistical analysis reveals, most notably, that: functional and scripting languages are more concise than procedural and object-oriented languages; C is hard to beat when it comes to raw speed on large inputs, but performance differences over inputs of moderate size are less pronounced and allow even interpreted languages to be competitive; compiled strongly-typed languages, where more defects can be caught at compile time, are less prone to runtime failures than interpreted or weakly-typed languages. We discuss implications of these results for developers, language designers, and educators.

I. INTRODUCTION

What is the best programming language for...? Questions about programming languages and the properties of their programs are asked often but well-founded answers are not easily available. From an engineering viewpoint, the design of a programming language is the result of multiple trade-offs that achieve certain desirable properties (such as speed) at the expense of others (such as simplicity). Technical aspects are, however, hardly ever the only relevant concerns when it comes to choosing a programming language. Factors as heterogeneous as a strong supporting community, similarity to other widespread languages, or availability of libraries are often instrumental in deciding a language's popularity and how it is used in the wild [16]. If we want to reliably answer questions about properties of programming languages, we have to analyze, empirically, the artifacts programmers write in those languages. Answers grounded in empirical evidence can be valuable in helping language users and designers make informed choices.

To control for the many factors that may affect the properties of programs, some empirical studies of programming languages [9], [21], [24], [30] have performed *controlled experiments* where human subjects (typically students) in highly controlled environments solve small programming tasks in different languages. Such controlled experiments provide the most reliable data about the impact of certain programming language features such as syntax and typing, but they are also necessarily limited in scope and generalizability by the number and types of tasks solved, and by the use of novice programmers as subjects. Real-world programming also develops over far more time than that allotted for short exam-like programming assignments; and produces programs that change features and improve quality over multiple development iterations.

At the opposite end of the spectrum, empirical studies based on analyzing programs in *public repositories* such as GitHub [2], [22], [25] can count on large amounts of mature code improved by experienced developers over substantial time spans. Such set-ups are suitable for studies of defect proneness and code evolution, but they also greatly complicate analyses that require directly comparable data across different languages: projects in code repositories target disparate categories of software, and even those in the same category (such as "web browsers") often differ broadly in features, design, and style, and hence cannot be considered to be implementing minor variants of the same task.

The study presented in this paper explores a middle ground between highly controlled but small programming assignments and large but incomparable software projects: programs in Rosetta Code. The Rosetta Code repository [27] collects solutions, written in hundreds of different languages, to an open collection of over 700 programming *tasks*. Most tasks are quite detailed descriptions of problems that go beyond simple programming assignments, from sorting algorithms to pattern matching and from numerical analysis to GUI programming. Solutions to the same task in different languages are thus significant samples of what each programming language can achieve and are directly comparable. The community of contributors to Rosetta Code (nearly 25'000 users at the time of writing) includes programmers that scrutinize, revise, and improve each other's solutions.

Our study analyzes 7'087 solution programs to 745 tasks in 8 widely used languages representing the major programming paradigms (procedural: C and Go; object-oriented: C# and Java; functional: F# and Haskell; scripting: Python and Ruby). The study's research questions target various program features including conciseness, size of executables, running time, memory usage, and failure proneness. A quantitative statistical analysis, cross-checked for consistency against a careful inspection of plotted data, reveals the following main findings about the programming languages we analyzed:

- Functional and scripting languages enable writing more concise code than procedural and object-oriented languages.
- Languages that compile into bytecode produce smaller executables than those that compile into native machine code.

- C is hard to beat when it comes to raw speed on large inputs. Go is the runner-up, also in terms of economical memory usage.
- In contrast, performance differences between languages shrink over inputs of moderate size, where languages with a lightweight runtime may be competitive even if they are interpreted.
- Compiled strongly-typed languages, where more defects can be caught at compile time, are less prone to runtime failures than interpreted or weakly-typed languages.

Beyond the findings specific to the programs and programming languages that we analyzed, our study paves the way for follow-up research that can benefit from the wealth of data in Rosetta Code and generalize our findings to other domains. To this end, Section IV discusses some practical implications of these findings for developers, language designers, and educators, whose choices about programming languages can increasingly rely on a growing fact base built on complementary sources.

The bulk of the paper describes the design of our empirical study (Section II), and its research questions and overall results (Section III). We refer to a detailed technical report [18] for the complete fine-grain details of the measures, statistics, and plots. To support repetition and replication studies, we also make the complete data available online[1], together with the scripts we wrote to produce and analyze it.

II. METHODOLOGY

A. The Rosetta Code repository

Rosetta Code [27] is a code repository with a wiki interface. This study is based on a repository's snapshot taken on 24 June 2014[2]; henceforth "Rosetta Code" denotes this snapshot.

Rosetta Code is organized in 745 *tasks*. Each task is a natural language description of a computational problem or theme, such as the bubble sort algorithm or reading the JSON data format. Contributors can provide *solutions* to tasks in their favorite programming languages, or revise already available solutions. Rosetta Code features 379 languages (with at least one solution per language) for a total of 49'305 solutions and 3'513'262 lines (total lines of program files). A solution consists of a piece of code, which ideally should accurately follow a task's description and be self-contained (including test inputs); that is, the code should compile and execute in a proper environment without modifications.

B. Task selection

Our study requires comparable solutions to clearly-defined tasks. To identify them, we categorized tasks, based on their description, according to whether they are suitable for lines-of-code analysis (LOC), compilation (COMP), and execution (EXEC); T_C denotes the set of tasks in a category C. Categories are increasingly restrictive: lines-of-code analysis only includes tasks sufficiently well-defined that their solutions can be considered minor variants of a unique problem; compilation further requires that tasks demand complete solutions rather

[1] https://bitbucket.org/nanzs/rosettacodedata
[2] Cloned into our Git repository[1] using a modified version of the Perl module RosettaCode-0.0.5 available from http://cpan.org/.

than sketches or snippets; execution further requires that tasks include meaningful inputs and algorithmic components (typically, as opposed to data-structure and interface definitions). As Table 1 shows, many tasks are too vague to be used in the study, but the differences between the tasks in the three categories are limited.

Table 1: CLASSIFICATION AND SELECTION OF ROSETTA CODE TASKS.

	ALL	LOC	COMP	EXEC	PERF	SCAL
# TASKS	745	454	452	436	50	46

Most tasks do not describe sufficiently precise and varied inputs to be usable in an analysis of runtime performance. For instance, some tasks are computationally trivial, and hence do not determine measurable resource usage when running. To identify tasks that can be meaningfully used in analyses of performance, we introduced two additional categories (PERF and SCAL) of tasks suitable for performance comparisons: PERF describes "everyday" workloads that are not necessarily very resource intensive, but whose descriptions include well-defined inputs that can be consistently used in every solution; in contrast, SCAL describes "computing-intensive" workloads with inputs that can easily be scaled up to substantial size and require well-engineered solutions. The corresponding sets T_{PERF} and T_{SCAL} are disjoint subsets of the execution tasks T_{EXEC}; Table 1 gives their size.

C. Language selection

Rosetta Code includes solutions in 379 languages. Analyzing all of them is not worth the huge effort, given that many languages are not used in practice or cover only few tasks. To find a representative and significant subset, we rank languages according to a combination of their rankings in Rosetta Code and in the TIOBE index [32]. A language's Rosetta Code ranking is based on the number of tasks for which at least one solution in that language exists: the larger the number of tasks the higher the ranking; Table 2 lists the top-20 languages (LANG) in the Rosetta Code ranking (ROSETTA) with the number of tasks they implement (# TASKS). The TIOBE programming community index [32] is a long-standing, monthly-published language popularity ranking based on hits in various search engines; Table 3 lists the top-20 languages in the TIOBE index with their TIOBE score (TIOBE).

A language ℓ must satisfy two criteria to be included in our study:

C1. ℓ ranks in the top-50 positions in the TIOBE index;
C2. ℓ implements at least one third (≈ 250) of the Rosetta Code tasks.

Criterion C1 selects widely-used, popular languages. Criterion C2 selects languages that can be compared on a substantial number of tasks, conducing to statistically significant results. Languages in Table 2 that fulfill criterion C1 are shaded (the top-20 in TIOBE are in bold); and so are languages in Table 3 that fulfill criterion C2.

Twenty-four languages satisfy both criteria. We assign scores to them, based on the following rules:

R1. A language ℓ receives a TIOBE score $\tau_\ell = 1$ iff it is in the top-20 in TIOBE (Table 3); otherwise, $\tau_\ell = 2$.

2

Table 2: Rosetta Code ranking: top 20.

ROSETTA	LANG	# TASKS	TIOBE
#1	Tcl	718	#43
#2	Racket	706	–[3]
#3	**Python**	675	#8
#4	Perl 6	644	–
#5	**Ruby**	635	#14
#6	J	630	–
#7	**C**	630	#1
#8	D	622	#50
#9	Go	617	#30
#10	PicoLisp	605	
#11	**Perl**	601	#11
#12	Ada	582	#29
#13	Mathematica	580	–
#14	REXX	566	–
#15	Haskell	553	#38
#16	AutoHotkey	536	–
#17	**Java**	534	#2
#18	BBC BASIC	515	–
#19	Icon	473	–
#20	OCaml	471	–

Table 3: TIOBE index ranking: top 20.

TIOBE	LANG	# TASKS	ROSETTA
#1	**C**	630	#7
#2	**Java**	534	#17
#3	**Objective-C**	136	#72
#4	**C++**	461	#22
#5	**(Visual) Basic**	34	#145
#6	**C#**	463	#21
#7	**PHP**	324	#36
#8	**Python**	675	#3
#9	**JavaScript**	371	#28
#10	**Transact-SQL**	4	#266
#11	**Perl**	601	#11
#12	**Visual Basic .NET**	104	#81
#13	**F#**	341	#33
#14	**Ruby**	635	#5
#15	**ActionScript**	113	#77
#16	**Swift**	–[4]	–
#17	**Delphi/Object Pascal**	219	#53
#18	**Lisp**	–[5]	–
#19	**MATLAB**	305	#40
#20	**Assembly**	–[5]	–

R2. A language ℓ receives a Rosetta Code score ρ_ℓ corresponding to its ranking in Rosetta Code (first column in Table 2).

Using these scores, languages are ranked in increasing lexicographic order of (τ_ℓ, ρ_ℓ). This ranking method sticks to the same rationale as C1 (prefer popular languages) and C2 (ensure a statistically significant base for analysis), and helps mitigate the role played by languages that are "hyped" in either the TIOBE or the Rosetta Code ranking.

To cover the most popular programming paradigms, we partition languages in four categories: procedural, object-oriented, functional, scripting. Two languages (R and MAT-LAB) mainly are special-purpose; hence we drop them. In each category, we rank languages using our ranking method and pick the top two languages. Table 4 shows the overall ranking; the shaded rows contain the eight languages selected for the study.

[3]No rank means that the language is not in the top-50 in the TIOBE index.
[4]Not represented in Rosetta Code.
[5]Only represented in Rosetta Code in dialect versions.

Table 4: Combined ranking: the top-2 languages in each category are selected for the study.

PROCEDURAL		OBJECT-ORIENTED		FUNCTIONAL		SCRIPTING	
ℓ	(τ_ℓ, ρ_ℓ)	ℓ	(τ_ℓ, ρ_ℓ)	ℓ	(τ_ℓ, ρ_ℓ)	ℓ	(τ_ℓ, ρ_ℓ)
C	(1,7)	**Java**	(1,17)	**F#**	(1,8)	**Python**	(1,3)
Go	(2,9)	**C#**	(1,21)	Haskell	(2,15)	**Ruby**	(1,5)
Ada	(2,12)	**C++**	(1,22)	Common Lisp	(2,23)	**Perl**	(1,11)
PL/I	(2,30)	D	(2,50)	Scala	(2,25)	**JavaScript**	(1,28)
Fortran	(2,39)			Erlang	(2,26)	**PHP**	(1,36)
				Scheme	(2,47)	Tcl	(2,1)
						Lua	(2,35)

D. Experimental setup

Rosetta Code collects solution files by task and language. The following table details the total size of the data considered in our experiments (LINES are total lines of program files).

	C	C#	F#	Go	Haskell	Java	Python	Ruby	ALL
TASKS	630	463	341	617	553	534	675	635	745
FILES	989	640	426	869	980	837	1'319	1'027	7'087
LINES	44'643	21'295	6'473	36'395	14'426	27'891	27'223	19'419	197'765

Our experiments measure properties of Rosetta Code solutions in various dimensions. We perform the following actions for each solution file f of every task t in each language ℓ:

- **Merge:** if f depends on other files (for example, an application consisting of two classes in two different files), make them available in the same location where f is; F denotes the resulting self-contained collection of source files that correspond to one solution of t in ℓ.
- **Patch:** if F has errors that prevent correct compilation or execution (for example, a library is used but not imported), correct F as needed.
- **LOC:** measure source-code features of F.
- **Compile:** compile F into native code (C, Go, and Haskell) or bytecode (C#, F#, Java, Python); *executable* denotes the files produced by compilation.[6] Measure compilation features.
- **Run:** run the executable and measure runtime features.

Merge. We stored the information necessary for this step in the form of makefiles—one for every task that requires merging, that is such that there is no one-to-one correspondence between source-code files and solutions. We wrote the makefiles after attempting a compilation with default options for all solution files, each compiled in isolation: we inspected all failed compilation attempts and provided makefiles whenever necessary.

Patch. We stored the information necessary for this step in the form of diffs—one for every solution file that requires correction. We wrote the diffs after attempting a compilation with the makefiles.

An important decision was *what to patch*. We want to have as many compiled solutions as possible, but we also do not want to alter the Rosetta Code data before measuring it. We did not fix errors that had to do with functional correctness or very solution-specific features. We did fix simple errors: missing library inclusions, omitted variable declarations, and typos. These guidelines try to replicate the moves of a user who would like to reuse Rosetta Code solutions but may not

[6]For Ruby, which does not produce compiled code of any kind, this step is replaced by a syntax check of F.

be fluent with the languages. As a result of following this process, we have a reasonably high confidence that patched solutions are correct implementations of the tasks.

Diffs play an additional role for tasks for performance analysis (T_{PERF} and T_{SCAL} in Section II-B). Solutions to these tasks must not only be correct but also run on the same inputs (tasks T_{PERF}) and on the same "large" inputs (tasks T_{SCAL}). We checked all solutions to tasks T_{PERF} and patched them when necessary to ensure they work on comparable inputs, but we did not change the inputs themselves from those suggested in the task descriptions. In contrast, we inspected all solutions to tasks T_{SCAL} and patched them by supplying task-specific inputs that are computationally demanding.

LOC. For each language ℓ, we wrote a script ℓ_loc that inputs a list of files, calls `cloc`[7] on them to count the lines of code, and logs the results.

Compile. For each language ℓ, we wrote a script ℓ_compile that inputs a list of files and compilation flags, calls the appropriate compiler on them, and logs the results. The following table shows the compiler versions used for each language, as well as the optimization flags. We tried to select a stable compiler version complete with matching standard libraries, and the best optimization level among those that are not too aggressive or involve rigid or extreme trade-offs.

LANG	COMPILER	VERSION	FLAGS
C	gcc (GNU)	4.6.3	-O2
C#	mcs (Mono 3.2.1)	3.2.1.0	-optimize
F#	fsharpc (Mono 3.2.1)	3.1	-O
Go	go	1.3	
Haskell	ghc	7.4.1	-O2
Java	javac (OracleJDK 8)	1.8.0_11	
Python	python (CPython)	2.7.3/3.2.3	
Ruby	ruby	2.1.2	-c

C_compile tries to detect the C dialect (gnu90, C99, ...) until compilation succeeds. Java_compile looks for names of public classes in each source file and renames the files to match the class names (as required by the Java compiler). Python_compile tries to detect the version of Python (2.x or 3.x) until compilation succeeds. Ruby_compile only performs a syntax check of the source (flag -c), since Ruby has no (standard) stand-alone compilation.

Run. For each language ℓ, we wrote a script ℓ_run that inputs an executable name, executes it, and logs the results. Native executables are executed directly, whereas bytecode is executed using the appropriate virtual machines. To have reliable performance measurements, the scripts: repeat each execution 6 times; discard the timing of the first execution (to fairly accommodate bytecode languages that load virtual machines from disk); check that the 5 remaining execution times are within one standard deviation of the mean; log the mean execution time. If an execution does not terminate within a time-out of 3 minutes it is forcefully terminated.

Overall process. For every language ℓ, for every task t, for each action $act \in \{$loc, compile, run$\}$:

1) if patches exist for any solution of t in ℓ, apply them;
2) if no makefile exists for task t in ℓ, call script ℓ_act directly on each solution file f of t;

---

[7] http://cloc.sourceforge.net/

---

3) if a makefile exists, invoke it and pass ℓ_act as command to be used; the makefile defines the self-contained collection of source files F on which the script works.

E. Experiments

The experiments ran on a Ubuntu 12.04 LTS 64bit GNU/Linux box with Intel Quad Core2 CPU at 2.40 GHz and 4 GB of RAM. At the end of the experiments, we extracted all logged data for statistical analysis using R.

F. Statistical analysis

The statistical analysis targets pairwise comparisons between languages. Each comparison uses a different metric M such as lines of code or CPU time. Let ℓ be a programming language, t a task, and M a metric. $\ell_M(t)$ denotes the vector of measures of M, one for each solution to task t in language ℓ. $\ell_M(t)$ may be empty if there are no solutions to task t in ℓ. The comparison of languages X and Y based on M works as follows. Consider a subset T of the tasks such that, for every $t \in T$, both X and Y have at least one solution to t.

Following this procedure, each T determines two data vectors x_M^α and y_M^α, for the two languages X and Y, by aggregating the measures per task using an *aggregation function* α; as aggregation functions, we normally consider both minimum and mean. For each task $t \in T$, the t-th component of the two vectors x_M^α and y_M^α is:

$$x_M^\alpha(t) = \alpha(X_M(t))/\nu_M(t, X, Y),$$
$$y_M^\alpha(t) = \alpha(Y_M(t))/\nu_M(t, X, Y),$$

where $\nu_M(t, X, Y)$ is a normalization factor defined as:

$$\nu_M(t, X, Y) = \begin{cases} \min(X_M(t)Y_M(t)) & \text{if } \min(X_M(t)Y_M(t)) > 0, \\ 1 & \text{otherwise}, \end{cases}$$

where juxtaposing vectors denotes concatenating them. This definition ensures that $x_M^\alpha(t)$ and $y_M^\alpha(t)$ are well-defined even when a minimum of zero occurs due to the limited precision of some measures such as running time.

As statistical test, we normally[8] use the Wilcoxon signed-rank test, a paired non-parametric difference test which assesses whether the mean ranks of x_M^α and of y_M^α differ. We display the test results in a table, under column labeled with language X at row labeled with language Y, and include various measures:

1) The p-value, which estimates the probability that chance can explain the differences between x_M^α and y_M^α. Intuitively, if p is small it means that there is a high chance that X and Y exhibit a genuinely different behavior with respect to metric M.
2) The effect size, computed as Cohen's d, defined as the standardized mean difference: $d = (\overline{x_M^\alpha} - \overline{y_M^\alpha})/s$, where $\overline{V}$ is the mean of a vector V, and s is the pooled standard deviation of the data. For statistically significant differences, d estimates how large the difference is.
3) The signed ratio

$$R = \text{sgn}(\widetilde{x_M^\alpha} - \widetilde{y_M^\alpha}) \frac{\max(\widetilde{x_M^\alpha}, \widetilde{y_M^\alpha})}{\min(\widetilde{x_M^\alpha}, \widetilde{y_M^\alpha})}$$

---

[8] Failure analysis (RQ5) uses the U test, as described there.

of the largest median to the smallest median (where $\widetilde{V}$ is the median of a vector V), which gives an unstandardized measure of the difference between the two medians.[9] Sign and absolute value of R have direct interpretations whenever the difference between X and Y is significant: if M is such that "smaller is better" (for instance, running time), then a *positive* sign $\mathrm{sgn}(\widetilde{x_M^\alpha} - \widetilde{y_M^\alpha})$ indicates that the average solution in language Y is *better* (smaller) with respect to M than the average solution in language X; the absolute value of R indicates *how many times X is larger than Y on average*.

Throughout the paper, we will say that language X: *is significantly different* from language Y, if $p < 0.01$; and that it *tends to be different* from Y if $0.01 \leq p < 0.05$. We will say that the effect size is: *vanishing* if $d < 0.05$; *small* if $0.05 \leq d < 0.3$; *medium* if $0.3 \leq d < 0.7$; and *large* if $d \geq 0.7$.

G. Visualizations of language comparisons

Each results table is accompanied by a *language relationship graph*, which helps visualize the results of the the pairwise language relationships. In such graphs, nodes correspond to programming languages. Two nodes ℓ_1 and ℓ_2 are arranged so that their *horizontal* distance is *roughly* proportional to the absolute value of ratio R for the two languages; an exact proportional display is not possible in general, as the pairwise ordering of languages may not be a total order. Vertical distances are chosen only to improve readability and carry no meaning.

A *solid* arrow is drawn from node X to Y if language Y is significantly better than language X in the given metric, and a *dashed* arrow if Y tends to be better than X (using the terminology from Section II-F). To improve the visual layout, edges that express an ordered pair that is subsumed by others are omitted, that is if $X \to W \to Y$ the edge from X to Y is omitted. The thickness of arrows is proportional to the effect size; if the effect is vanishing, no arrow is drawn.

III. RESULTS

RQ1. Which programming languages make for more concise code?

To answer this question, we measure the non-blank non-comment *lines of code* of solutions of tasks T_{LOC} marked for lines of code count that compile without errors. The requirement of successful compilation ensures that only syntactically correct programs are considered to measure conciseness. To check the impact of this requirement, we also compared these results with a measurement including all solutions (whether they compile or not), obtaining qualitatively similar results.

For all research questions but RQ5, we considered both minimum and mean as aggregation functions (Section II-F). For brevity, the presentation describes results for only one of them (typically the minimum). For lines of code measurements, aggregating by minimum means that we consider, for each task, the shortest solution available in the language.

Table 5 shows the results of the pairwise comparison, where p is the p-value, d the effect size, and R the ratio, as described in Section II-F. In the table, ε denotes the smallest positive floating-point value representable in R.

Table 5: COMPARISON OF LINES OF CODE (BY MINIMUM).

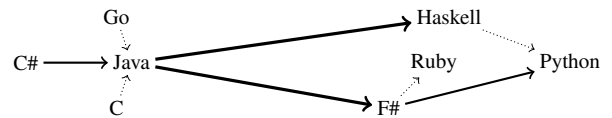| LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	0.543						
	d	0.004						
	R	-1.1						
F#	p	$<\varepsilon$	$<\varepsilon$					
	d	0.735	0.945					
	R	2.5	2.6					
Go	p	0.377	0.082	$<10^{-29}$				
	d	0.155	0.083	0.640				
	R	1.0	1.0	-2.5				
Haskell	p	$<\varepsilon$	$<\varepsilon$	0.168	$<\varepsilon$			
	d	1.071	1.286	0.085	1.255			
	R	2.9	2.7	1.3	2.9			
Java	p	0.026	$<10^{-4}$	$<10^{-25}$	0.026	$<10^{-32}$		
	d	0.262	0.319	0.753	0.148	1.052		
	R	1.0	1.1	-2.3	1.0	-2.9		
Python	p	$<\varepsilon$	$<\varepsilon$	$<10^{-4}$	$<\varepsilon$	0.021	$<\varepsilon$	
	d	0.951	1.114	0.359	0.816	0.209	0.938	
	R	2.9	3.6	1.6	3.0	1.2	2.8	
Ruby	p	$<\varepsilon$	$<\varepsilon$	0.013	$<\varepsilon$	0.764	$<\varepsilon$	0.015
	d	0.558	0.882	0.103	0.742	0.107	0.763	0.020
	R	2.5	2.7	1.2	2.5	-1.2	2.2	-1.3

Figure 6: Comparison of lines of code (by minimum).

Figure 6 shows the corresponding language relationship graph; remember that arrows point to the more concise languages, thickness denotes larger effects, and horizontal distances are roughly proportional to average differences. Languages are clearly divided into two groups: functional and scripting languages tend to provide the most concise code, whereas procedural and object-oriented languages are significantly more verbose. The absolute difference between the two groups is major; for instance, Java programs are on average 2.2–2.9 times longer than programs in functional and scripting languages.

Within the two groups, differences are less pronounced. Among the scripting languages, and among the functional languages, no statistically significant differences exist. Python tends to be the most concise, even against functional languages (1.2–1.6 times shorter on average). Among procedural and object-oriented languages, Java tends to be slightly more concise, with small to medium effect sizes.

> *Functional and scripting languages provide significantly more concise code than procedural and object-oriented languages.*

RQ2. Which programming languages compile into smaller executables?

To answer this question, we measure the *size of the executables* of solutions of tasks T_{COMP} marked for compilation that compile without errors. We consider both native-code executables (C, Go, and Haskell) and bytecode executables (C#, F#, Java, Python). Ruby's standard programming environment does not offer compilation to bytecode and Ruby programs are

therefore not included in the measurements for RQ2.

Table 7 shows the results of the statistical analysis, and Figure 8 the corresponding language relationship graph.

Table 7: COMPARISON OF SIZE OF EXECUTABLES (BY MINIMUM).

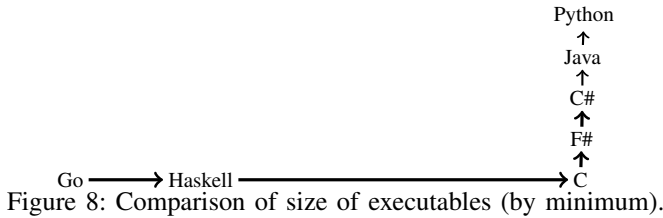| LANG		C	C#	F#	Go	Haskell	Java
C#	p	$<\varepsilon$					
	d	2.669					
	R	2.1					
F#	p	$<\varepsilon$	$<10^{-15}$				
	d	1.395	1.267				
	R	1.4	-1.5				
Go	p	$<10^{-52}$	$<10^{-39}$	$<10^{-31}$			
	d	3.639	2.312	2.403			
	R	-153.3	-340.7	-217.8			
Haskell	p	$<10^{-45}$	$<10^{-35}$	$<10^{-29}$	$<\varepsilon$		
	d	2.469	2.224	2.544	1.071		
	R	-111.2	-240.3	-150.8	1.3		
Java	p	$<\varepsilon$	$<10^{-4}$	$<\varepsilon$	$<\varepsilon$	$<\varepsilon$	
	d	3.148	0.364	1.680	3.121	1.591	
	R	2.3	1.1	1.7	341.4	263.8	
Python	p	$<\varepsilon$	$<10^{-15}$	$<\varepsilon$	$<\varepsilon$	$<\varepsilon$	$<10^{-5}$
	d	5.686	0.899	1.517	3.430	1.676	0.395
	R	2.9	1.3	2.0	452.7	338.3	1.3

Figure 8: Comparison of size of executables (by minimum).

It is apparent that measuring executable sizes determines a total order of languages, with Go producing the largest and Python the smallest executables. Based on this order, three consecutive groups naturally emerge: Go and Haskell compile to native and have "large" executables; F#, C#, Java, and Python compile to bytecode and have "small" executables; C compiles to native but with size close to bytecode executables.

Size of bytecode does not differ much across languages: F#, C#, and Java executables are, on average, only 1.3–2.0 times larger than Python's. The differences between sizes of native executables are more spectacular, with Go's and Haskell's being on average 153.3 and 111.2 times larger than C's. This is largely a result of Go and Haskell using static linking by default, as opposed to gcc defaulting to dynamic linking whenever possible. With dynamic linking, C produces very compact binaries, which are on average a mere 2.9 times larger than Python's bytecode. C was compiled with level -O2 optimization, which should be a reasonable middle ground: binaries tend to be larger under more aggressive speed optimizations, and smaller under executable size optimizations (flag -Os).

> *Languages that compile into bytecode have significantly smaller executables than those that compile into native machine code.*

RQ3. Which programming languages have better running-time performance?

To answer this question, we measure the *running time* of solutions of tasks T_{SCAL} marked for running time measurements on computing-intensive workloads that run without errors or timeout (set to 3 minutes). As discussed in Section II-B

and Section II-D, we manually patched solutions to tasks in T_{SCAL} to ensure that they work on the same inputs of substantial size. This ensures that—as is crucial for running time measurements—all solutions used in these experiments run on the very same inputs.

Table 9: COMPUTING-INTENSIVE TASKS.

	NAME	INPUT
1	9 billion names of God the integer	$n = 10^5$
2–3	Anagrams	$100 \times$ unixdict.txt (20.6 MB)
4	Arbitrary-precision integers	$5^{4^{3^2}}$
5	Combinations	$\binom{25}{10}$
6	Count in factors	$n = 10^6$
7	Cut a rectangle	10×10 rectangle
8	Extensible prime generator	10^7th prime
9	Find largest left truncatable prime	10^7th prime
10	Hamming numbers	10^7th Hamming number
11	Happy numbers	10^6th Happy number
12	Hofstadter Q sequence	# flips up to 10^5th term
13–16	Knapsack problem/[*all versions*]	from task description
17	Ludic numbers	from task description
18	LZW compression	$100 \times$ unixdict.txt (20.6 MB)
19	Man or boy test	$n = 16$
20	N-queens problem	$n = 13$
21	Perfect numbers	first 5 perfect numbers
22	Pythagorean triples	perimeter $< 10^8$
23	Self-referential sequence	$n = 10^6$
24	Semordnilap	$100 \times$ unixdict.txt
25	Sequence of non-squares	non-squares $< 10^6$
26–34	Sorting algorithms/[*quadratic*]	$n \simeq 10^4$
35–41	Sorting algorithms/[$n \log n$ *and linear*]	$n \simeq 10^6$
42–43	Text processing/[*all versions*]	from task description (1.2 MB)
44	Topswops	$n = 12$
45	Towers of Hanoi	$n = 25$
46	Vampire number	from task description

Table 9 summarizes the tasks T_{SCAL} and their inputs. It is a diverse collection which spans from text processing tasks on large input files ("Anagrams", "Semordnilap"), to combinatorial puzzles ("N-queens problem", "Towers of Hanoi"), to NP-complete problems ("Knapsack problem") and sorting algorithms of varying complexity. We chose inputs sufficiently large to probe the performance of the programs, and to make input/output overhead negligible w.r.t. total running time. Table 10 shows the results of the statistical analysis, and Figure 11 the corresponding language relationship graph.

Table 10: COMPARISON OF RUNNING TIME (BY MINIMUM) FOR COMPUTING-INTENSIVE TASKS.

| LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	0.001						
	d	0.328						
	R	-7.5						
F#	p	0.012	0.075					
	d	0.453	0.650					
	R	-8.6	-2.6					
Go	p	$<10^{-4}$	0.020	0.016				
	d	0.453	0.338	0.578				
	R	-1.6	6.3	5.6				
Haskell	p	$<10^{-4}$	0.084	0.929	$<10^{-3}$			
	d	0.895	0.208	0.424	0.705			
	R	-24.5	-2.9	-1.6	-13.1			
Java	p	$<10^{-4}$	0.661	0.158	0.0135	0.098		
	d	0.374	0.364	0.469	0.563	0.424		
	R	-3.2	1.8	4.9	-2.4	6.6		
Python	p	$<10^{-5}$	0.033	0.938	$<10^{-3}$	0.894	0.082	
	d	0.704	0.336	0.318	0.703	0.386	0.182	
	R	-29.8	-4.3	-1.1	-15.5	1.1	-9.2	
Ruby	p	$<10^{-3}$	0.004	0.754	$<10^{-3}$	0.360	0.013	0.055
	d	0.999	0.358	0.113	0.984	0.250	0.204	0.020
	R	-34.1	-6.2	-1.4	-18.5	-1.4	-7.9	-1.6
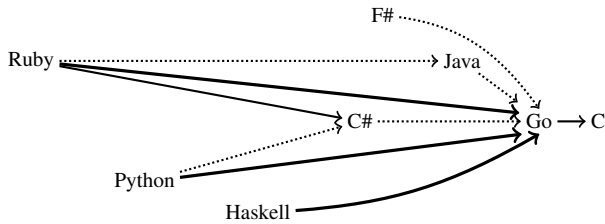
Figure 11: Comparison of running time (by minimum) for computing-intensive tasks.

C is unchallenged over the computing-intensive tasks T_{SCAL}. Go is the runner-up, but significantly slower with medium effect size: the average Go program is 1.6 times slower than the average C program. Programs in other languages are much slower than Go programs, with medium to large effect size (2.4–18.5 times slower than Go on average).

> *C is king on computing-intensive workloads. Go is the runner-up. Other languages, with object-oriented or functional features, incur further performance losses.*

The results on the tasks T_{SCAL} clearly identified the procedural languages—C in particular—as the fastest. However, the raw speed demonstrated on those tasks represents challenging conditions that are relatively infrequent in the many classes of applications that are not algorithmically intensive. To find out performance differences under other conditions, we measure running time on the tasks T_{PERF}, which are still clearly defined and run on the same inputs, but are not markedly computationally intensive and do not naturally scale to large instances. Examples of such tasks are checksum algorithms (Luhn's credit card validation), string manipulation tasks (reversing the space-separated words in a string), and standard system library accesses (securing a temporary file).

The results, which we only discuss in the text for brevity, are definitely more mixed than those related to tasks T_{SCAL}, which is what one could expect given that we are now looking into modest running times in absolute value, where every language has at least decent performance. First of all, C loses its undisputed supremacy, as it is not significantly faster than Go and Haskell—even though, when differences are statistically significant, C remains ahead of the other languages. The procedural languages and Haskell collectively emerge as the fastest in tasks T_{PERF}; none of them sticks out as *the* fastest because the differences among them are insignificant and may sensitively depend on the tasks that each language implements in Rosetta Code. Among the other languages (C#, F#, Java, Python, and Ruby), Python emerges as the fastest. Overall, we confirm that the distinction between T_{PERF} and T_{SCAL} tasks—which we dub "everyday" and "computing-intensive"—is quite important to understand performance differences among languages. On tasks T_{PERF}, languages with an agile runtime, such as the scripting languages, or with natively efficient operations on lists and string, such as Haskell, may turn out to be efficient in practice.

> *The distinction between "everyday" and "computing-intensive" workloads is important when assessing running-time performance. On "everyday" workloads, languages may be able to compete successfully regardless of their programming paradigm.*

RQ4. Which programming languages use memory more efficiently?

To answer this question, we measure the maximum RAM usage (i.e., maximum resident set size) of solutions of tasks T_{SCAL} marked for comparison on computing-intensive tasks that run without errors or timeout; this measure includes the memory footprint of the runtime environments. Table 12 shows the results of the statistical analysis, and Figure 13 the corresponding language relationship graph.

Table 12: COMPARISON OF MAXIMUM RAM USED (BY MINIMUM).

| LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	<10-4						
	d	2.022						
	R	-2.5						
F#	p	0.006	0.010					
	d	0.761	1.045					
	R	-4.5	-5.2					
Go	p	<10-3	<10-4	0.006				
	d	0.064	0.391	0.788				
	R	-1.8	4.1	3.5				
Haskell	p	<10-3	0.841	0.062	<10-3			
	d	0.287	0.123	0.614	0.314			
	R	-14.2	-3.5	2.5	-5.6			
Java	p	<10-5	<10-4	0.331	<10-5	0.007		
	d	0.890	1.427	0.278	0.527	0.617		
	R	-5.4	-2.4	1.4	-2.6	2.1		
Python	p	<10-5	0.351	0.034	<10-4	0.992	0.006	
	d	0.330	0.445	0.096	0.417	0.010	0.206	
	R	-5.0	-2.2	1.0	-2.7	2.9	-1.1	
Ruby	p	<10-5	0.002	0.530	<10-4	0.049	0.222	0.036
	d	0.403	0.525	0.242	0.531	0.301	0.301	0.061
	R	-5.0	-5.0	1.5	-2.5	1.8	-1.1	1.0
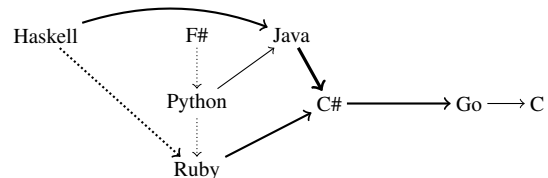
Figure 13: Comparison of maximum RAM used (by minimum).

C and Go clearly emerge as the languages that make the most economical usage of RAM. Go's frugal memory usage (on average only 1.8 times higher than C) is remarkable, given that its runtime includes garbage collection. In contrast, all other languages use considerably more memory (2.5–14.2 times on average over either C or Go), which is justifiable in light of their bulkier runtimes, supporting not only garbage collection but also features such as dynamic binding (C# and Java), lazy evaluation, pattern matching (Haskell and F#), dynamic typing, and reflection (Python and Ruby).

Differences between languages in the same category (procedural, scripting, and functional) are generally small or insignificant. The exception are object-oriented languages, where Java uses significantly more RAM than C# (on average, 2.4 times more). Among the other languages, Haskell emerges as the least memory-efficient, although some differences are insignificant.

While maximum RAM usage is a major indication of the efficiency of memory usage, modern architectures include many-layered memory hierarchies whose influence on performance is multi-faceted. To complement the data about maximum RAM and refine our understanding of memory

usage, we also measured *average* RAM usage and number of *page faults*. Average RAM tends to be practically zero in all tasks but very few; correspondingly, the statistics are inconclusive as they are based on tiny samples. By contrast, the data about page faults clearly partitions the languages in two classes: the functional languages trigger significantly more page faults than all other languages; in fact, the only statistically significant differences are those involving F# or Haskell, whereas programs in other languages hardly ever trigger a single page fault. The difference in page faults between Haskell programs and F# programs is insignificant. The page faults recorded in our experiments indicate that functional languages exhibit significant non-locality of reference. The overall impact of this phenomenon probably depends on a machine's architecture; RQ3, however, showed that functional languages are generally competitive in terms of running-time performance, so that their non-local behavior might just denote a particular instance of the space vs. time trade-off.

> *Procedural languages use significantly less memory than other languages. Functional languages make distinctly non-local memory accesses.*

RQ5. Which programming languages are less failure prone?

To answer this question, we measure *runtime failures* of solutions of tasks T_{EXEC} marked for execution that compile without errors or timeout. We exclude programs that time out because whether a timeout is indicative of failure depends on the task: for example, interactive applications will time out in our setup waiting for user input, but this should not be recorded as failure. Thus, a terminating program *fails* if it returns an exit code other than 0 (for example, when throwing an uncaught exception). The measure of failures is ordinal and not normalized: ℓ_f denotes a vector of binary values, one for each solution in language ℓ where we measure runtime failures; a value in ℓ_f is 1 iff the corresponding program fails and it is 0 if it does not fail.

Data about failures differs from that used to answer the other research questions in that we cannot aggregate it by task, since failures in different solutions, even for the same task, are in general unrelated. Therefore, we use the Mann-Whitney U test, an unpaired non-parametric ordinal test which can be applied to compare samples of different size. For two languages X and Y, the U test assesses whether the two samples X_f and Y_f of binary values representing failures are likely to come from the same population.

Table 14: NUMBER OF SOLUTIONS THAT RAN WITHOUT TIMEOUT, AND THEIR PERCENTAGE THAT RAN WITHOUT ERRORS.

	C	C#	F#	Go	Haskell	Java	Python	Ruby
# ran solutions	391	246	215	389	376	297	675	516
% no error	87%	93%	89%	98%	93%	85%	85%	86%

Table 15 shows the results of the tests; we do not report unstandardized measures of difference, such as R in the previous tables, since they would be uninformative on ordinal data. Figure 16 is the corresponding language relationship graph. Horizontal distances are proportional to the fraction of solutions that run without errors (last row of Table 14).

Go clearly sticks out as the least failure prone language. Other data suggests that the number of compile-time errors

Table 15: COMPARISONS OF RUNTIME FAILURE PRONENESS.

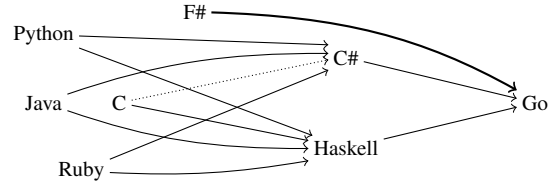| LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	0.037						
	d	0.170						
F#	p	0.500	0.204					
	d	0.057	0.119					
Go	p	$<10^{-7}$	0.011	$<10^{-5}$				
	d	0.410	0.267	0.398				
Haskell	p	0.006	0.748	0.083	0.002			
	d	0.200	0.026	0.148	0.227			
Java	p	0.386	0.006	0.173	$<10^{-9}$	$<10^{-3}$		
	d	0.067	0.237	0.122	0.496	0.271		
Python	p	0.332	0.003	0.141	$<10^{-10}$	$<10^{-3}$	0.952	
	d	0.062	0.222	0.115	0.428	0.250	0.004	
Ruby	p	0.589	0.010	0.260	$<10^{-9}$	$<10^{-3}$	0.678	0.658
	d	0.036	0.201	0.091	0.423	0.231	0.030	0.026

Figure 16: Comparisons of runtime failure proneness.

in Go programs is similar to the other compiled languages'. Together, these two elements indicate that the Go compiler is particularly good at catching sources of failures at compile time, since only a small fraction of compiled programs fail at runtime. Go's restricted type system (no inheritance, no overloading, no genericity, no pointer arithmetic) may help make compile-time checks effective. By contrast, the scripting languages tend to be the most failure prone of the lot. This is a consequence of Python and Ruby being *interpreted* languages[10]: any syntactically correct program is executed, and hence most errors manifest themselves only at runtime.

There are few major differences among the remaining *compiled* languages, where it is useful to distinguish between weak (C) and strong (the other languages) type systems [8, Sec. 3.4.2]. F# shows no statistically significant differences with any of C, C#, and Haskell. C tends to be more failure prone than C# and is significantly more failure prone than Haskell; similarly to the explanation behind the interpreted languages' failure proneness, C's weak type system may be responsible for fewer failures being caught at compile time than at runtime. In fact, the association between weak typing and failure proneness was also found in other studies [25]. Java is unusual in that it has a strong type system and is compiled, but is significantly more error prone than Haskell and C#, which also are strongly typed and compiled. Future work will determine if Java's behavior is spurious or indicative of concrete issues.

> *Compiled strongly-typed languages are significantly less prone to runtime failures than interpreted or weakly-typed languages, since more errors are caught at compile time. Thanks to its simple static type system, Go is the least failure-prone language in our study.*

---

[10]Even if Python compiles to bytecode, the translation process only performs syntactic checks (and is not invoked separately normally anyway).

IV. Discussion

The results of our study can help different stakeholders—developers, language designers, and educators—to make better informed choices about language usage and design.

The conciseness of functional and scripting programming languages suggests that the characterizing features of these languages—such as list comprehensions, type polymorphism, dynamic typing, and extensive support for reflection and list and map data structures—provide for great expressiveness. In times where more and more languages combine elements belonging to different paradigms, language designers can focus on these features to improve the expressiveness and raise the level of abstraction. For programmers, using a programming language that makes for concise code can help write software with fewer bugs. In fact, some classic research suggests [11], [14], [15] that bug density is largely constant across programming languages—all else being equal [7], [17]; therefore, shorter programs will tend to have fewer bugs.

The results about executable size are an instance of the ubiquitous space vs. time trade-off. Languages that compile to native can perform more aggressive compile-time optimizations since they produce code that is very close to the actual hardware it will be executed on. However, with the ever increasing availability of cheap and compact memory, differences between languages have significant implications only for applications that run on highly constrained hardware such as embedded devices. Interpreted languages such as Ruby exercise yet another trade-off, where there is no visible binary at all and all optimizations are done at runtime.

No one will be surprised by our results that C dominates other languages in terms of raw speed and efficient memory usage. Major progresses in compiler technology notwithstanding, higher-level programming languages do incur a noticeable performance loss to accommodate features such as automatic memory management or dynamic typing in their runtimes. Nevertheless, our results on "everyday" workloads showed that pretty much any language can be competitive when it comes to the regular-size inputs that make up the overwhelming majority of programs. When teaching and developing software, we should then remember that "most applications do not actually need better performance than Python offers" [26, p. 337].

Another interesting lesson emerging from our performance measurements is how Go achieves respectable running times as well as good results in memory usage, thereby distinguishing itself from the pack just as C does (in fact, Go's developers include prominent figures who were also primarily involved in the development of C). Go's design choices may be traced back to a careful selection of features that differentiates it from most other language designs (which tend to be more feature-prodigal): while it offers automatic memory management and some dynamic typing, it deliberately omits genericity and inheritance, and offers only a limited support for exceptions. In our study, we have seen that Go features not only good performance but also a compiler that is quite effective at finding errors at compile time rather than leaving them to leak into runtime failures. Besides being appealing for certain kinds of software development (Go's concurrency mechanisms, which we didn't consider in this study, may be another feature to consider), Go also shows to language designers that there still

is uncharted territory in the programming language landscape.

Evidence in our, as well as others' (Section VI), analysis tends to confirm what advocates of static strong typing have long claimed: that it makes it possible to catch more errors earlier, at compile time. But the question remains of which process leads to overall higher programmer productivity (or, in a different context, to effective learning): postponing testing and catching as many errors as possible at compile time, or running a prototype as soon as possible while frequently going back to fixing and refactoring? The traditional knowledge that bugs are more expensive to fix the later they are detected is not an argument against the "test early" approach, since testing early may be the quickest way to find an error in the first place. This is another area where new trade-offs can be explored by selectively—or flexibly [1]—combining features.

V. Threats to Validity

Threats to *construct validity*—are we measuring the right things?—are quite limited given that our research questions, and the measures we take to answer them, target widespread well-defined features (conciseness, performance, and so on) with straightforward matching measures (lines of code, running time, and so on). A partial exception is RQ5, which targets the multifaceted notion of failure proneness, but the question and its answer are consistent with related empirical work that approached the same theme from other angles, which reflects positively on the soundness of our constructs. Regarding conciseness, lines of code remains a widely used metric, but it will be interesting to correlate it with other proposed measures of conciseness.

We took great care in the study's design and execution to minimize threats to *internal validity*—are we measuring things right? We manually inspected all task descriptions to ensure that the study only includes well-defined tasks and comparable solutions. We also manually inspected, and modified whenever necessary, all solutions used to measure performance, where it is of paramount importance that the same inputs be applied in every case. To ensure reliable runtime measures (running time, memory usage, and so on), we ran every executable multiple times, checked that each repeated run's deviation from the average is moderate (less than one standard deviation), and based our statistics on the average (mean) behavior. Data analysis often showed highly statistically significant results, which also reflects favorably on the soundness of the study's data. Our experimental setup tried to use *default* settings; this may limit the scope of our findings, but also helps reduce bias. Exploring different directions, such as pursuing the best optimizations possible in each language [21], is an interesting goal of future work.

A possible threat to *external validity*—do the findings generalize?—has to do with whether the properties of Rosetta Code programs are representative of real-world software projects. On one hand, Rosetta Code tasks tend to favor algorithmic problems, and solutions are quite small on average compared to any realistic application or library. On the other hand, every large project is likely to include a small set of core functionalities whose quality, performance, and reliability significantly influences the whole system's; Rosetta Code programs can be indicative of such core functionalities. In

addition, measures of performance are meaningful only on comparable implementations of algorithmic tasks, and hence Rosetta Code's algorithmic bias helped provide a solid base for comparison of this aspect (Section II-B and RQ3,4). The size and level of activity of the Rosetta Code community mitigates the threat that contributors to Rosetta Code are not representative of the skills of real-world programmers. However, to ensure wider generalizability, we plan to analyze other characteristics of the Rosetta Code programming community.

Another potential threat comes from the choice of programming languages. Section II-C describes how we selected languages representative of real-world popularity among major paradigms. Classifying programming languages into paradigms has become harder in recent times, when multi-paradigm languages are the norm. Nonetheless, we maintain that paradigms still significantly influence the way in which programs are written, and it is natural to associate major programming languages to a specific paradigm based on their Rosetta Code programs. For example, even though Python offers classes and other object-oriented features, practically no solutions in Rosetta Code use them. Extending the study to more languages and new paradigms belongs to future work.

VI. RELATED WORK

Controlled experiments. Prechelt [24] compares 7 programming languages on a single task in 80 solutions written by students. Findings include: the program written in Perl, Python, REXX, or Tcl is "only half as long" as written in C, C++, or Java; C and C++ are generally faster than Java. The study asks questions similar to ours but is limited by the small sample size. Languages and their compilers have evolved since 2000 (when [24] was published), making the results difficult to compare; however, some tendencies (conciseness of scripting languages, performance-dominance of C) are visible in our study too. Harrison et al. [10] compare the code quality of C++ against SML's on 12 tasks, finding few significant differences. Our study targets a broader set of research questions (only RQ5 is related to quality). Hanenberg [9] conducts a study with 49 students over 27 hours of development time comparing static vs. dynamic type systems, finding no significant differences. In contrast to controlled experiments, our approach cannot take development time into account.

Studying 15 students on a single problem, Szafron and Schaeffer [31] identify a message-passing library that is somewhat superior to higher-level parallel programming, even though the latter is more "usable" overall. This highlights the difficulty of reconciling results of different metrics. We do not attempt this in our study, as the suitability of a language for certain projects may depend on external factors. Other studies [4], [5], [12], [13] compare parallel programming approaches (UPC, MPI, OpenMP, and X10) using mostly small student populations. In the realm of concurrent programming, a study [28] with 237 undergraduate students implementing one program with locks, monitors, or transactions suggests that transactions leads to the fewest errors. In a usability study with 67 students [19], we find advantages of the SCOOP concurrency model over Java's monitors. Pankratius et al. [23] compare Scala and Java using 13 students and one software engineer working on three tasks. They conclude that Scala's functional style leads to more compact code and comparable performance. To eschew the limitations of classroom studies—based on the unrepresentative performance of novice programmers—previous work of ours [20], [21] compared Chapel, Cilk, Go, and TBB on 96 solutions to 6 tasks that were checked for style and performance by language experts.

A common problem with all the aforementioned studies is that they often target few tasks and solutions, and therefore fail to achieve statistical significance or generalizability. The large sample size in our study minimizes these problems.

Surveys. Meyerovich and Rabkin [16] study the reasons behind language adoption. One key finding is that the intrinsic features of a language (such as reliability) are less important for adoption when compared to extrinsic ones such as existing code. This puts our study into perspective, and shows that some features we investigate are very important to developers (e.g., performance). Bissyandé et al. [3] study similar questions. Their rankings, according to lines of code or usage in projects, may suggest alternatives to the TIOBE ranking we used.

Repository mining. Bhattacharya and Neamtiu [2] study 4 projects in C and C++ to understand the impact on software quality, finding an advantage in C++. With similar goals, Ray et al. [25] mine 729 projects in 17 languages from GitHub. They find that strong typing is modestly better than weak typing, and functional languages have an advantage over procedural languages. Our study looks at a broader spectrum of research questions in a more controlled environment, but our results on failures (RQ5) confirm the superiority of statically strongly typed languages. Other studies investigate specialized features of programming languages. For example, recent studies by us [6] and others [29] investigate the use of contracts and their interplay with other language features such as inheritance. Okur and Dig [22] analyze 655 open-source applications with parallel programming to identify adoption trends and usage problems, addressing questions that are orthogonal to ours.

VII. CONCLUSIONS

Programming languages are essential tools for the working computer scientist, and it is no surprise that what is the "right tool for the job" can be the subject of intense debates. To put such debates on strong foundations, we must understand how features of different languages relate to each other. Our study revealed differences regarding some of the most frequently discussed language features—conciseness, performance, failure-proneness—and is therefore of value to software developers and language designers. The key to having highly significant statistical results in our study was the use of a large program chrestomathy: Rosetta Code. The repository can be a valuable resource also for future programming language research that corroborates, or otherwise complements, our findings.

REFERENCES

[1] M. Bayne, R. Cook, and M. D. Ernst, "Always-available static and dynamic feedback," in *Proceedings of the 33rd International Conference*

[2] P. Bhattacharya and I. Neamtiu, "Assessing programming language impact on development and maintenance: A study on C and C++," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 171–180.

[3] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, ser. COMPSAC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 303–312.

[4] F. Cantonnet, Y. Yao, M. M. Zahran, and T. A. El-Ghazawi, "Productivity analysis of the UPC language," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, ser. IPDPS '04. Los Alamitos, CA, USA: IEEE Computer Society, 2004.

[5] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, and J. Urbanic, "An experiment in measuring the productivity of three parallel programming languages," in *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*, ser. P-PHEC '06, 2006, pp. 30–37.

[6] H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer, "Contracts in practice," in *Proceedings of the 19th International Symposium on Formal Methods (FM)*, ser. Lecture Notes in Computer Science, vol. 8442. Springer, 2014, pp. 230–246.

[7] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[8] C. Ghezzi and M. Jazayeri, *Programming language concepts*, 3rd ed. Wiley & Sons, 1997.

[9] S. Hanenberg, "An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 22–35.

[10] R. Harrison, L. G. Samaraweera, M. R. Dobie, and P. H. Lewis, "Comparing programming paradigms: an evaluation of functional and object-oriented programs," *Software Engineering Journal*, vol. 11, no. 4, pp. 247–254, July 1996.

[11] L. Hatton, "Computer programming languages and safety-related systems," in *Proceedings of the 3rd Safety-Critical Systems Symposium*. Berlin, Heidelberg: Springer, 1995, pp. 182–196.

[12] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert, "A pilot study to compare programming effort for two parallel programming models," *Journal of Systems and Software*, vol. 81, pp. 1920–1930, 2008.

[13] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel programmer productivity: A case study of novice parallel programmers," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 35–43.

[14] C. Jones, *Programming Productivity*. Mcgraw-Hill College, 1986.

[15] S. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.

[16] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 1–18.

[17] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 282–292.

[18] S. Nanz and C. A. Furia, "A comparative study of programming languages in Rosetta Code," http://arxiv.org/abs/1409.0252, September 2014.

[19] S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer, "Design of an empirical study for comparing the usability of concurrent programming languages," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 325–334.

on *Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 521–530.

[20] S. Nanz, S. West, and K. Soares da Silveira, "Examining the expert gap in parallel programming," in *Proceedings of the 19th European Conference on Parallel Processing (Euro-Par '13)*, ser. Lecture Notes in Computer Science, vol. 8097. Berlin, Heidelberg: Springer, 2013, pp. 434–445.

[21] S. Nanz, S. West, K. Soares da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," in *Proceedings of the 7th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 183–192.

[22] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 54:1–54:11.

[23] V. Pankratius, F. Schmidt, and G. Garretón, "Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE '12. IEEE, 2012, pp. 123–133.

[24] L. Prechelt, "An empirical comparison of seven programming languages," *IEEE Computer*, vol. 33, no. 10, pp. 23–29, Oct. 2000.

[25] B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2014.

[26] E. S. Raymond, *The Art of UNIX Programming*. Addison-Wesley, 2003.

[27] Rosetta Code, June 2014. [Online]. Available: http://rosettacode.org/

[28] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 47–56.

[29] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst, "Case studies and tools for contract specifications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 596–607.

[30] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[31] D. Szafron and J. Schaeffer, "An experiment to measure the usability of parallel programming systems," *Concurrency: Practice and Experience*, vol. 8, no. 2, pp. 147–166, 1996.

[32] TIOBE Programming Community Index, July 2014. [Online]. Available: http://www.tiobe.com