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

arXiv:1409.0252v2 [cs.SE] 5 Sep 2014

***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, to draw a fair and well-founded comparison. Rosetta Code offers a large data set for analysis. Our study is based on 7087 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, who can make better informed choices about programming languages.**

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 [15]. 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 [8], [19], [22], [28] 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], [20], [23] 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 [25] 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. At the same time, the community of contributors to Rosetta Code (nearly 25'000 users at the time of writing) includes expert programmers that scrutinize and revise each other's solutions; this makes for programs of generally high quality which are representative of proper usage of the languages by experts.

Our study analyzes 7087 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, and makes a particularly frugal usage of memory.
- In contrast, performance differences between languages shrink over inputs of moderate size, where languages with a lightweight runtime may have an edge 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.

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 [16] 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 [25] 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.

Tasks significantly differ in the detail, prescriptiveness, and generality of their descriptions. The most detailed ones, such as "Bubble sort", consist of well-defined algorithms, described informally and in pseudo-code, and include tests (input/output pairs) to demonstrate solutions. Other tasks are much vaguer and only give a general theme, which may be inapplicable to some languages or admit widely different solutions. For instance, task "Memory allocation" just asks to "show how to explicitly allocate and deallocate blocks of memory".

B. Task selection

Whereas even vague task descriptions may prompt well-written solutions, 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 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.

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

Table 1: Classification and selection of Rosetta Code tasks.

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; others do not give specific inputs to be tested, and hence solutions may run on incomparable inputs; others still are well-defined but their performance without interactive input is immaterial, such as in the case of graphic animation tasks. 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. For example, sorting algorithms are computing-intensive tasks working on large input lists; "Cholesky matrix decomposition" is an everyday performance task working on two test input matrices that can be decomposed quickly. 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 [30]. 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 [30] 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:

[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/.

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 2: Rosetta Code 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]	–

Table 3: TIOBE index ranking: top 20.

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. A comparison of the two tables indicates that some popular languages are underrepresented in Rosetta Code, such as Objective-C, (Visual) Basic, and Transact-SQL; conversely, some languages popular in Rosetta Code have a low TIOBE ranking, such as Tcl, Racket, and Perl 6.

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, $\tau_\ell = 2$.
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 MATLAB) 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.

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)

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

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: source-code features (such as lines of code), compilation features (such as size of executables), and runtime features (such as execution time). Correspondingly, we have to 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*

[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.

denotes the files produced by compilation.[6] Measure compilation features.

- **Run:** run the executable and measure runtime features.

Actions *merge* and *patch* are solution-specific and are required for the actions that follow. In contrast, *LOC*, *compile*, and *run* are only language-specific and produce the actual experimental data. To automate executing the actions to the extent possible, we built a system of scripts that we now describe in some detail.

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. A makefile has one target for every task solution F, and a default `all` target that builds all solution targets for the current task. Each target's recipe calls a placeholder script `comp`, passing to it the list of input files that constitute the solution together with other necessary solution-specific compilation files (for example, library flags for the linker). 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: we inspected all failed compilation attempts, and wrote diffs whenever necessary. Some corrections could not be expressed as diffs because they involved renaming or splitting files (for example, some C files include both declarations and definitions, but the former should go in separate header files); we implemented these corrections by adding shell commands directly in 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 be fluent with the languages. In general, the quality of Rosetta Code solutions is quite high, and hence we have a reasonably high confidence that all patched solutions are indeed 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 (everyday tasks T_{PERF}) and on the same "large" inputs (computing-intensive tasks T_{SCAL}). We checked all solutions to performance tasks 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. A significant example of computing-intensive tasks were the sorting algorithms, which we patched to build and sort large integer arrays (generated on the fly using a linear congruential generator function with fixed seed). The input size was chosen after a few trials so as to be feasible for most languages within a timeout of 3 minutes; for example, the sorting algorithms deal with arrays of size from $3 \cdot 10^4$ elements for quadratic-time algorithms to $2 \cdot 10^6$ elements for linear-time algorithms.

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; the timing of the first execution is discarded (to fairly accommodate bytecode languages that load virtual machines from disk: it is only in the first execution that the virtual machine is loaded from disk, with corresponding possibly significant one-time overhead; in the successive executions the virtual machine is read from cache, with only limited overhead). If an execution does not terminate within a time-out of 3 minutes it is forcefully terminated.

Overall process. A Python script orchestrates the whole experiment. 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;
3) if a makefile exists, invoke it and pass ℓ`_act` as command `comp`to be used; the makefile defines the self-contained collection of source files F on which the script works.

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

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

Since the command-line interface of the ℓ_`loc`, ℓ_`compile`, and ℓ_`run` scripts is uniform, the same makefiles work as recipes for all actions `act`.

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 including lines of code (conciseness), size of the executable (native or bytecode), CPU time, maximum RAM usage (i.e., maximum resident set size), number of page faults, and number of runtime failures. Metrics are *normalized* as we detail below. 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. T may be further restricted based on a measure-dependent *criterion*; for example, to check conciseness, we may choose to only consider a task t if both X and Y have at least one solution that compiles without errors (solutions that do not satisfy the criterion are discarded).

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. Thus, the normalization factor is the smallest value of metric M measured across all solutions of t in X and in Y if such a value is positive; otherwise, when the minimum is zero, the normalization factor is one. 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 the differences between x_M^α and y_M^α are due to chance. If p is small it means that there is a high chance that X and Y exhibit a genuinely different behavior w.r.t. 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

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

of the largest mean to the smallest mean, which gives an unstandardized measure of the difference between the two means. 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 $\mathrm{sgn}(\overline{x_M^\alpha} - \overline{y_M^\alpha})$ 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 $0.01 \le p < 0.05$. We will say that the effect size is: *vanishing* if $d < 0.05$; *small* if $0.05 \le d < 0.3$; *medium* if $0.3 \le d < 0.7$; and *large* if $d \ge 0.7$.

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

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

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.

LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	0.543						
	d	0.004						
	R	1.0						
F#	p	$<\varepsilon$	$<\varepsilon$					
	d	0.735	0.945					
	R	4.9	4.1					
Go	p	0.377	0.082	$<10^{-29}$				
	d	0.155	0.083	0.640				
	R	1.1	1.1	-4.5				
Haskell	p	$<\varepsilon$	$<\varepsilon$	0.168	$<\varepsilon$			
	d	1.071	1.286	0.085	1.255			
	R	3.8	3.7	1.1	3.5			
Java	p	0.026	$<10^{-4}$	$<10^{-25}$	0.026	$<10^{-32}$		
	d	0.262	0.319	0.753	0.148	1.052		
	R	1.1	1.2	-3.6	1.1	-3.4		
Python	p	$<\varepsilon$	$<\varepsilon$	$<10^{-4}$	$<\varepsilon$	0.021	$<\varepsilon$	
	d	0.951	1.114	0.359	0.816	0.209	0.938	
	R	4.5	4.8	1.3	4.5	1.2	3.9	
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	5.2	4.8	1.1	4.6	1.1	3.9	1.0

Table 5: Comparison of lines of code (by minimum).

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 3.4–3.9 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. Functional programs tend to be more verbose than scripts, although only with small to medium effect sizes (1.1–1.3 times larger on average). Among procedural and object-oriented languages, Java tends to be more concise: C, C#, and Go programs are 1.1–1.2 times larger than Java programs on average, corresponding to 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) as well as 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.

LANG		C	C#	F#	Go	Haskell	Java
C#	p	$<\varepsilon$					
	d	2.669					
	R	2.4					
F#	p	$<\varepsilon$	$<10^{-15}$				
	d	1.395	1.267				
	R	1.6	-1.6				
Go	p	$<10^{-52}$	$<10^{-39}$	$<10^{-31}$			
	d	3.639	2.312	2.403			
	R	-154.3	-387.0	-257.9			
Haskell	p	$<10^{-45}$	$<10^{-35}$	$<10^{-29}$	$<\varepsilon$		
	d	2.469	2.224	2.544	1.071		
	R	-110.4	-267.3	-173.6	1.4		
Java	p	$<\varepsilon$	$<10^{-4}$	$<\varepsilon$	$<\varepsilon$	$<\varepsilon$	
	d	3.148	0.364	1.680	3.121	1.591	
	R	2.7	1.2	1.8	414.6	313.1	
Python	p	$<\varepsilon$	$<10^{-15}$	$<\varepsilon$	$<\varepsilon$	$<\varepsilon$	$<10^{-5}$
	d	5.686	0.899	1.517	3.430	1.676	0.395
	R	3.0	1.4	2.1	475.7	352.9	1.3

Table 7: Comparison of size of executables (by minimum).

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, two consecutive groups naturally emerge: Go, Haskell, and C compile to native and have "large" executables; and F#, C#, Java, and Python compile to bytecode and have "small" executables.

Size of bytecode does not differ much across languages: F#, C#, and Java executables are, on average, only 1.3–2.1 times larger than Python's. The differences between sizes of native executables is more spectacular, with Go's and Haskell's being on average 154.3 and 110.4 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 3 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.

	NAME	INPUT
1	9 billion names of God the integer	$n = 10^5$
2–3	Anagrams	100 × `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 × `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 × `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: Computing-intensive tasks.

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.

C is unchallenged over the computing-intensive tasks T_{SCAL}. Go is the runner-up but still significantly slower with medium effect size: the average Go program is 18.7 times slower than the average C program. Programs in other languages are much slower than Go programs, with medium to large effect size (4.6–13.7 times slower than Go on average).

LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	0.001						
	d	0.328						
	R	-63.2						
F#	p	0.012	0.075					
	d	0.453	0.650					
	R	-94.5	-4.0					
Go	p	$<10^{-4}$	0.020	0.016				
	d	0.453	0.338	0.578				
	R	-18.7	6.6	13.7				
Haskell	p	$<10^{-4}$	0.084	0.929	$<10^{-3}$			
	d	0.895	0.208	0.424	0.705			
	R	-64.4	2.8	29.0	-13.6			
Java	p	$<10^{-4}$	0.661	0.158	0.0135	0.098		
	d	0.374	0.364	0.469	0.563	0.424		
	R	-33.7	-10.5	14.0	-4.6	8.7		
Python	p	$<10^{-5}$	0.027	0.938	$<10^{-3}$	0.877	0.079	
	d	0.711	0.336	0.318	0.709	0.408	0.116	
	R	-42.3	-27.8	-2.2	-9.8	5.7	1.7	
Ruby	p	$<10^{-3}$	0.004	0.754	$<10^{-3}$	0.360	0.013	0.071
	d	0.999	0.358	0.113	0.984	0.250	0.204	0.019
	R	-8.6	-11.6	1.4	-9.7	4.0	2.6	-1.1

Table 10: Comparison of running time (by minimum) for computing-intensive tasks.

Figure 11: Comparison of running time (by minimum) for computing-intensive tasks.

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

The results on the computing-intensive 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 on everyday programs, 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 computing-intensive workloads, 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 absolute supremacy, as it is significantly slower than Python, Ruby, and Haskell—even though the effect sizes are smallish, and C remains ahead of the other languages. The scripting 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 small and may sensitively depend on the tasks that each language implements

in Rosetta Code. There is also no language among the others (C#, F#, Go, and Java) that clearly emerges as the fastest, even though some differences are significant. Overall, we confirm that the distinction between "everyday" and "computing-intensive" tasks 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 the most 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. Table 12 shows the results of the statistical analysis, and Figure 13 the corresponding language relationship graph.

LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	$<10^{-4}$						
	d	2.022						
	R	-8.4						
F#	p	0.006	0.010					
	d	0.761	1.045					
	R	-22.4	-4.1					
Go	p	$<10^{-3}$	$<10^{-4}$	0.006				
	d	0.064	0.391	0.788				
	R	1.2	20.4	10.5				
Haskell	p	$<10^{-3}$	0.841	0.062	$<10^{-3}$			
	d	0.287	0.123	0.614	0.314			
	R	-18.1	-1.2	7.4	-18.6			
Java	p	$<10^{-5}$	$<10^{-4}$	0.331	$<10^{-5}$	0.007		
	d	0.890	1.427	0.278	0.527	0.617		
	R	-41.4	-3.4	-1.143	-35.0	-3.2		
Python	p	$<10^{-5}$	0.351	0.0342	$<10^{-4}$	0.992	0.006	
	d	0.330	0.445	0.104	0.417	0.009	0.202	
	R	-25.6	-3.2	-1.4	-17.9	1.0	1.5	
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	0.064
	R	-44.8	-6.1	1.2	-26.1	-2.5	1.7	1.3

Table 12: Comparison of maximum RAM used (by minimum).

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 is even significantly more frugal than C—a remarkable feature given that Go's runtime includes garbage collection—although the magnitude of its advantage is small (C's maximum RAM usage is on average 1.2 times higher). In contrast, all other languages use considerably more memory (8.4–44.8 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 (object-oriented, scripting, and functional) are generally small or insignificant. The exception is Java, which uses significantly more RAM than C#, Haskell, and Python; the average difference, however, is comparatively small (1.5–3.4 times on average). Comparisons between languages in different categories are also mixed or inconclusive: the scripting languages tend to use more RAM than Haskell, and Python tends to use more RAM than F#, but the difference between F# and Ruby is insignificant; C# uses significantly less RAM than F#, but Haskell uses less RAM than Java, and other differences between object-oriented and functional languages 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. Then, F# programs cause fewer page faults than Haskell programs on average, although the difference is borderline significant ($p \approx 0.055$). 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, with Go being the most frugal even with automatic memory management. 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. 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.

	C	C#	F#	Go	Haskell	Java	Python	Ruby
# ran solutions	391	246	215	389	376	297	676	516
% no error	87%	93%	89%	98%	93%	85%	79%	86%

Table 14: Number of solutions that ran without timeout, and their percentage that ran without errors.

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).

LANG		C	C#	F#	Go	Haskell	Java	Python
C#	p	0.037						
	d	0.170						
F#	p	0.500	0.200					
	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	$<10^{-3}$	$<10^{-5}$	$<10^{-3}$	$<10^{-16}$	$<10^{-8}$	0.030	
	d	0.215	0.360	0.260	0.558	0.393	0.151	
Ruby	p	0.589	0.010	0.260	$<10^{-9}$	$<10^{-3}$	0.678	0.002
	d	0.036	0.201	0.091	0.423	0.230	0.030	0.183

Table 15: Comparisons of runtime failure proneness.

Figure 16: Comparisons of runtime failure proneness.

	C	C#	F#	Go	Haskell	Java	Python	Ruby
# comp. solutions	524	354	254	497	519	446	775	581
% no error	85%	90%	95%	89%	84%	78%	100%	100%

Table 17: Number of solutions considered for compilation, and their percentage that compiled without errors.

Go clearly sticks out as the least failure prone language. If we look, in Table 17, at the fraction of solutions that failed to compile, and hence didn't contribute data to failure analysis, Go is not significantly different from 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) likely helps make compile-time checks effective. By contrast, the scripting languages tend to be the most failure prone of the lot; Python, in particular, is significantly more failure prone than every other language. This is a consequence of Python and Ruby being *interpreted* languages[9]: 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 [7, 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 is likely partly 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 [23]. 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. Our data suggests that the root cause for this phenomenon is in Java's choice of checking for the presence of a `main` method only at runtime upon invocation of the virtual machine on a specific compiled class. Whereas Haskell and C# programs without a `main` entry point fail to compile into an executable, Java's compile without errors but later trigger a runtime exception.

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.

IV. Implications

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, it is generally understood [10], [13], [14] that bug density is largely constant across programming languages all else being equal; 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. In fact, compilers to native tend to have several optimization options, which exercise different trade-offs. GNU's `gcc`, for instance, has a `-Os` flag that optimizes for executable size instead of speed (but we didn't use this highly specialized optimization in our

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

experiments). 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(where, in fact, bytecode languages are becoming increasingly common). Finally, 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. What is surprising is, perhaps, that C is still so widespread even for projects where maximum speed is hardly a requirement. 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" [24, p. 337].

Another interesting lesson emerging from our performance measurements is how Go achieves respectable running times as well as excellent results in memory usage, thereby distinguishing itself from the pack just as C does. It is no coincidence that Go's developers include prominent figures—Ken Thompson, most notably—who were also primarily involved in the development of C. The good performance of Go is a result of 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 this trade-off achieves 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, and innovative solutions could be discovered that are germane to requirements in certain special domains.

Evidence in our, as well as others' (Section VI), analysis confirms 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 what 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 featuresthat enhance compilation or execution.

V. Threats to Validity

Threats to *construct validity*—are we asking the right questions?—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.

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 negligible, 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 *standard* tools with *default* settings; this may limit the scope of our findings, but also helps reduce biasdue to different familiarity with different languages. Exploring different directions, such as pursuing the best optimizations possible in each language [19]for each task, 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 are 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). Finally, 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 and expertise of experienced programmers.

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(many programming languages offer procedures, some form of object system, and even functional features such as closures and list comprehensions).

[10]Nonetheless, we maintain that paradigms still significantly influence the typical "style" in which programs are written, and it is natural to associate major programming languages to a specific style based on their Rosetta Code programs. For example, even if 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 are a popular approach to language comparisons: study participants program the same tasks in different languages while researchers measure features such as code size and execution or development time. Prechelt [22] compares 7 programming languages on a single task in 80 solutions written by studentsand other volunteers. Measures include program size, execution time, memory consumption, and development time. Findings include: the program written in Perl, Python, REXX, or Tcl is "only half as long" as written in C, C++, or Java; performance results are more mixed, but 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, 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. [9] compare the code quality of C++ against the functional language 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 [8] 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.

Many recent comparative studies have targeted programming languages for concurrency and parallelism. Studying 15 students on a single problem, Szafron and Schaeffer [29] 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 factorsthat assign different weights to different metrics. Other studies [4], [5], [11], [12] compare parallel programming approaches (UPC, MPI, OpenMP, and X10) using mostly small student populations. In the realm of concurrent programming, a study [26] 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 [17], we find advantages of the SCOOP concurrency model over Java's monitors. Pankratius et al. [21] 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 (for instance, in [5], about a third of the student subjects fail the parallel programming task in that they cannot achieve any speedup)—previous work of ours [18], [19] compared Chapel, Cilk, Go, and TBB on 96 solutions to 6 tasks that were checked for style and performance by notable language experts. [18], [19] also introduced language dependency diagrams similar to those used in the present paper.

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 can help characterize the perception of programming languages. Meyerovich and Rabkin [15] 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, open-source libraries, and previous experience. This puts our study into perspective, and shows that some features we investigate are very important to developers (e.g., performanceas second most important attribute). Bissyandé et al. [3] study similar questions: the popularity, interoperability, and impact of languages. Their rankings, according to lines of code or usage in projects, may suggest alternatives to the TIOBE ranking we usedfor selecting languages.

Repository mining, as we have done in this study, has become a customary approach to answering a variety of questions about programming languages. 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. [23] 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 [27] investigate the use of contracts and their interplay with other language features such as inheritance. Okur and Dig [20] 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 language designers, as well as to developers choosing a language for their projects. 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. Besides using Rosetta Code,

[10]At the 2012 LASER summer school on "Innovative languages for software engineering", Mehdi Jazayeri mentioned the proliferation of multi-paradigm languages as a disincentive to updating his book on programming language concepts [7].

researchers can also improve it (by correcting any detected errors) and can increase its research value (by maintaining easily accessible up-to-date statistics).

Acknowledgments. Thanks to Rosetta Code's Mike Mol for helpful replies to our questions about the repository. We thank members of the Chair of Software Engineering for their helpful comments on a draft of this paper. This work was partially supported by ERC grant CME #291389.

References

[1] M. Bayne, R. Cook, and M. D. Ernst, "Always-available static and dynamic feedback," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 521–530.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

CONTENTS

I Introduction 1

II Methodology 2

II-A The Rosetta Code repository . . . 2

II-B Task selection . . . 2

II-C Language selection . . . 2

II-D Experimental setup . . . 3

II-E Experiments . . . 5

II-F Statistical analysis . . . 5

II-G Visualizations of language comparisons . . . 5

III Results 5

IV Implications 9

V Threats to Validity 10

VI Related Work 11

VII Conclusions 11

References 12

VIII Appendix: Pairwise comparisons 24

VIII-A Conciseness . . . 25

VIII-B Conciseness (all tasks) . . . 25

VIII-C Comments . . . 25

VIII-D Binary size . . . 26

VIII-E Performance . . . 26

VIII-F Scalability . . . 26

VIII-G Memory usage . . . 26

VIII-H Page faults . . . 26

VIII-I Timeouts . . . 26

VIII-J Solutions per task . . . 27

VIII-K Other comparisons . . . 27

VIII-L Compilation . . . 27

VIII-M Execution . . . 28

VIII-N Overall code quality (compilation + execution) . . . 29

VIII-O Fault proneness . . . 29

IX Appendix: Tables and graphs 30

IX-A Lines of code (tasks compiling successfully) . . . 30
IX-B Lines of code (all tasks) . . . 34
IX-C Comments per line of code . . . 38
IX-D Size of binaries . . . 42
IX-E Performance . . . 46
IX-F Scalability . . . 50
IX-G Maximum RAM . . . 54
IX-H Page faults . . . 58
IX-I Timeout analysis . . . 62
IX-J Number of solutions . . . 64
IX-K Compilation and execution statistics . . . 66

X Appendix: Plots 71

X-A Lines of code (tasks compiling successfully) . . . 72
X-B Lines of code (all tasks) . . . 97
X-C Comments per line of code . . . 122
X-D Size of binaries . . . 147
X-E Performance . . . 168
X-F Scalability . . . 193
X-G Maximum RAM . . . 218
X-H Page faults . . . 243
X-I Timeout analysis . . . 268
X-J Number of solutions . . . 281

LIST OF FIGURES

6 Comparison of lines of code (by minimum). . . . 6
8 Comparison of size of executables (by minimum). . . . 6
11 Comparison of running time (by minimum) for computing-intensive tasks. . . . 7
13 Comparison of maximum RAM used (by minimum). . . . 8
16 Comparisons of runtime failure proneness. . . . 9
20 Lines of code (min) of tasks compiling successfully . . . 31
21 Lines of code (min) of tasks compiling successfully (normalized horizontal distances) . . . 31
23 Lines of code (mean) of tasks compiling successfully . . . 33
24 Lines of code (mean) of tasks compiling successfully (normalized horizontal distances) . . . 33
26 Lines of code (min) of all tasks . . . 35
27 Lines of code (min) of all tasks (normalized horizontal distances) . . . 35
29 Lines of code (mean) of all tasks . . . 37
30 Lines of code (mean) of all tasks (normalized horizontal distances) . . . 37
32 Comments per line of code (min) of all tasks . . . 39
33 Comments per line of code (min) of all tasks (normalized horizontal distances) . . . 39

35 Comments per line of code (mean) of all tasks . . . 41
36 Comments per line of code (mean) of all tasks (normalized horizontal distances) . . . 41
38 Size of binaries (min) of tasks compiling successfully . . . 43
39 Size of binaries (min) of tasks compiling successfully (normalized horizontal distances) . . . 43
41 Size of binaries (mean) of tasks compiling successfully . . . 45
42 Size of binaries (mean) of tasks compiling successfully (normalized horizontal distances) . . . 45
44 Performance (min) of tasks running successfully . . . 47
45 Performance (min) of tasks running successfully (normalized horizontal distances) . . . 47
47 Performance (mean) of tasks running successfully . . . 49
48 Performance (mean) of tasks running successfully (normalized horizontal distances) . . . 49
50 Scalability (min) of tasks running successfully . . . 51
51 Scalability (min) of tasks running successfully (normalized horizontal distances) . . . 51
52 Scalability (min) of tasks running successfully (normalized horizontal distances) . . . 51
54 Scalability (mean) of tasks running successfully . . . 53
55 Scalability (mean) of tasks running successfully (normalized horizontal distances) . . . 53
56 Scalability (mean) of tasks running successfully (normalized horizontal distances) . . . 53
58 Maximum RAM usage (min) of scalability tasks . . . 55
59 Maximum RAM usage (min) of scalability tasks (normalized horizontal distances) . . . 55
60 Maximum RAM usage (min) of scalability tasks (normalized horizontal distances) . . . 55
62 Maximum RAM usage (mean) of scalability tasks . . . 57
63 Maximum RAM usage (mean) of scalability tasks (normalized horizontal distances) . . . 57
64 Maximum RAM usage (mean) of scalability tasks (normalized horizontal distances) . . . 57
66 Page faults (min) of scalability tasks . . . 59
67 Page faults (min) of scalability tasks (normalized horizontal distances) . . . 59
69 Page faults (mean) of scalability tasks . . . 61
70 Page faults (mean) of scalability tasks (normalized horizontal distances) . . . 61
72 Timeout analysis of scalability tasks . . . 63
73 Timeout analysis of scalability tasks (normalized horizontal distances) . . . 63
75 Number of solutions per task . . . 65
76 Number of solutions per task (normalized horizontal distances) . . . 65
78 Comparisons of compilation status . . . 66
80 Comparisons of running status . . . 67
82 Comparisons of combined compilation and running status . . . 68
84 Comparisons of fault proneness (based on exit status) of solutions that compile correctly and do not timeout . . . 69
88 Lines of code (min) of tasks compiling successfully (C vs. other languages) . . . 73
89 Lines of code (min) of tasks compiling successfully (C vs. other languages) . . . 74
90 Lines of code (min) of tasks compiling successfully (C# vs. other languages) . . . 75
91 Lines of code (min) of tasks compiling successfully (C# vs. other languages) . . . 76
92 Lines of code (min) of tasks compiling successfully (F# vs. other languages) . . . 77
93 Lines of code (min) of tasks compiling successfully (F# vs. other languages) . . . 78

94 Lines of code (min) of tasks compiling successfully (Go vs. other languages) . . . 79
95 Lines of code (min) of tasks compiling successfully (Go vs. other languages) . . . 80
96 Lines of code (min) of tasks compiling successfully (Haskell vs. other languages) . . . 81
97 Lines of code (min) of tasks compiling successfully (Haskell vs. other languages) . . . 82
98 Lines of code (min) of tasks compiling successfully (Java vs. other languages) . . . 82
99 Lines of code (min) of tasks compiling successfully (Java vs. other languages) . . . 83
100 Lines of code (min) of tasks compiling successfully (Python vs. other languages) . . . 83
101 Lines of code (min) of tasks compiling successfully (Python vs. other languages) . . . 83
102 Lines of code (min) of tasks compiling successfully (all languages) . . . 84
103 Lines of code (mean) of tasks compiling successfully (C vs. other languages) . . . 85
104 Lines of code (mean) of tasks compiling successfully (C vs. other languages) . . . 86
105 Lines of code (mean) of tasks compiling successfully (C# vs. other languages) . . . 87
106 Lines of code (mean) of tasks compiling successfully (C# vs. other languages) . . . 88
107 Lines of code (mean) of tasks compiling successfully (F# vs. other languages) . . . 89
108 Lines of code (mean) of tasks compiling successfully (F# vs. other languages) . . . 90
109 Lines of code (mean) of tasks compiling successfully (Go vs. other languages) . . . 91
110 Lines of code (mean) of tasks compiling successfully (Go vs. other languages) . . . 92
111 Lines of code (mean) of tasks compiling successfully (Haskell vs. other languages) . . . 93
112 Lines of code (mean) of tasks compiling successfully (Haskell vs. other languages) . . . 94
113 Lines of code (mean) of tasks compiling successfully (Java vs. other languages) . . . 94
114 Lines of code (mean) of tasks compiling successfully (Java vs. other languages) . . . 95
115 Lines of code (mean) of tasks compiling successfully (Python vs. other languages) . . . 95
116 Lines of code (mean) of tasks compiling successfully (Python vs. other languages) . . . 95
117 Lines of code (mean) of tasks compiling successfully (all languages) . . . 96
118 Lines of code (min) of all tasks (C vs. other languages) . . . 98
119 Lines of code (min) of all tasks (C vs. other languages) . . . 99
120 Lines of code (min) of all tasks (C# vs. other languages) . . . 100
121 Lines of code (min) of all tasks (C# vs. other languages) . . . 101
122 Lines of code (min) of all tasks (F# vs. other languages) . . . 102
123 Lines of code (min) of all tasks (F# vs. other languages) . . . 103
124 Lines of code (min) of all tasks (Go vs. other languages) . . . 104
125 Lines of code (min) of all tasks (Go vs. other languages) . . . 105
126 Lines of code (min) of all tasks (Haskell vs. other languages) . . . 106
127 Lines of code (min) of all tasks (Haskell vs. other languages) . . . 107
128 Lines of code (min) of all tasks (Java vs. other languages) . . . 107
129 Lines of code (min) of all tasks (Java vs. other languages) . . . 108
130 Lines of code (min) of all tasks (Python vs. other languages) . . . 108
131 Lines of code (min) of all tasks (Python vs. other languages) . . . 108
132 Lines of code (min) of all tasks (all languages) . . . 109
133 Lines of code (mean) of all tasks (C vs. other languages) . . . 110

134 Lines of code (mean) of all tasks (C vs. other languages) . . . 111
135 Lines of code (mean) of all tasks (C# vs. other languages) . . . 112
136 Lines of code (mean) of all tasks (C# vs. other languages) . . . 113
137 Lines of code (mean) of all tasks (F# vs. other languages) . . . 114
138 Lines of code (mean) of all tasks (F# vs. other languages) . . . 115
139 Lines of code (mean) of all tasks (Go vs. other languages) . . . 116
140 Lines of code (mean) of all tasks (Go vs. other languages) . . . 117
141 Lines of code (mean) of all tasks (Haskell vs. other languages) . . . 118
142 Lines of code (mean) of all tasks (Haskell vs. other languages) . . . 119
143 Lines of code (mean) of all tasks (Java vs. other languages) . . . 119
144 Lines of code (mean) of all tasks (Java vs. other languages) . . . 120
145 Lines of code (mean) of all tasks (Python vs. other languages) . . . 120
146 Lines of code (mean) of all tasks (Python vs. other languages) . . . 120
147 Lines of code (mean) of all tasks (all languages) . . . 121
148 Comments per line of code (min) of all tasks (C vs. other languages) . . . 123
149 Comments per line of code (min) of all tasks (C vs. other languages) . . . 124
150 Comments per line of code (min) of all tasks (C# vs. other languages) . . . 125
151 Comments per line of code (min) of all tasks (C# vs. other languages) . . . 126
152 Comments per line of code (min) of all tasks (F# vs. other languages) . . . 127
153 Comments per line of code (min) of all tasks (F# vs. other languages) . . . 128
154 Comments per line of code (min) of all tasks (Go vs. other languages) . . . 129
155 Comments per line of code (min) of all tasks (Go vs. other languages) . . . 130
156 Comments per line of code (min) of all tasks (Haskell vs. other languages) . . . 131
157 Comments per line of code (min) of all tasks (Haskell vs. other languages) . . . 132
158 Comments per line of code (min) of all tasks (Java vs. other languages) . . . 132
159 Comments per line of code (min) of all tasks (Java vs. other languages) . . . 133
160 Comments per line of code (min) of all tasks (Python vs. other languages) . . . 133
161 Comments per line of code (min) of all tasks (Python vs. other languages) . . . 133
162 Comments per line of code (min) of all tasks (all languages) . . . 134
163 Comments per line of code (mean) of all tasks (C vs. other languages) . . . 135
164 Comments per line of code (mean) of all tasks (C vs. other languages) . . . 136
165 Comments per line of code (mean) of all tasks (C# vs. other languages) . . . 137
166 Comments per line of code (mean) of all tasks (C# vs. other languages) . . . 138
167 Comments per line of code (mean) of all tasks (F# vs. other languages) . . . 139
168 Comments per line of code (mean) of all tasks (F# vs. other languages) . . . 140
169 Comments per line of code (mean) of all tasks (Go vs. other languages) . . . 141
170 Comments per line of code (mean) of all tasks (Go vs. other languages) . . . 142
171 Comments per line of code (mean) of all tasks (Haskell vs. other languages) . . . 143
172 Comments per line of code (mean) of all tasks (Haskell vs. other languages) . . . 144
173 Comments per line of code (mean) of all tasks (Java vs. other languages) . . . 144

174 Comments per line of code (mean) of all tasks (Java vs. other languages) . . . 145
175 Comments per line of code (mean) of all tasks (Python vs. other languages) . . . 145
176 Comments per line of code (mean) of all tasks (Python vs. other languages) . . . 145
177 Comments per line of code (mean) of all tasks (all languages) . . . 146
178 Size of binaries (min) of tasks compiling successfully (C vs. other languages) . . . 148
179 Size of binaries (min) of tasks compiling successfully (C vs. other languages) . . . 149
180 Size of binaries (min) of tasks compiling successfully (C# vs. other languages) . . . 150
181 Size of binaries (min) of tasks compiling successfully (C# vs. other languages) . . . 151
182 Size of binaries (min) of tasks compiling successfully (F# vs. other languages) . . . 152
183 Size of binaries (min) of tasks compiling successfully (F# vs. other languages) . . . 153
184 Size of binaries (min) of tasks compiling successfully (Go vs. other languages) . . . 154
185 Size of binaries (min) of tasks compiling successfully (Go vs. other languages) . . . 155
186 Size of binaries (min) of tasks compiling successfully (Haskell vs. other languages) . . . 155
187 Size of binaries (min) of tasks compiling successfully (Haskell vs. other languages) . . . 156
188 Size of binaries (min) of tasks compiling successfully (Java vs. other languages) . . . 156
189 Size of binaries (min) of tasks compiling successfully (Java vs. other languages) . . . 156
190 Size of binaries (min) of tasks compiling successfully (Python vs. other languages) . . . 157
191 Size of binaries (min) of tasks compiling successfully (Python vs. other languages) . . . 157
192 Size of binaries (min) of tasks compiling successfully (all languages) . . . 157
193 Size of binaries (mean) of tasks compiling successfully (C vs. other languages) . . . 158
194 Size of binaries (mean) of tasks compiling successfully (C vs. other languages) . . . 159
195 Size of binaries (mean) of tasks compiling successfully (C# vs. other languages) . . . 160
196 Size of binaries (mean) of tasks compiling successfully (C# vs. other languages) . . . 161
197 Size of binaries (mean) of tasks compiling successfully (F# vs. other languages) . . . 162
198 Size of binaries (mean) of tasks compiling successfully (F# vs. other languages) . . . 163
199 Size of binaries (mean) of tasks compiling successfully (Go vs. other languages) . . . 164
200 Size of binaries (mean) of tasks compiling successfully (Go vs. other languages) . . . 165
201 Size of binaries (mean) of tasks compiling successfully (Haskell vs. other languages) . . . 165
202 Size of binaries (mean) of tasks compiling successfully (Haskell vs. other languages) . . . 166
203 Size of binaries (mean) of tasks compiling successfully (Java vs. other languages) . . . 166
204 Size of binaries (mean) of tasks compiling successfully (Java vs. other languages) . . . 166
205 Size of binaries (mean) of tasks compiling successfully (Python vs. other languages) . . . 167
206 Size of binaries (mean) of tasks compiling successfully (Python vs. other languages) . . . 167
207 Size of binaries (mean) of tasks compiling successfully (all languages) . . . 167
208 Performance (min) of tasks running successfully (C vs. other languages) . . . 169
209 Performance (min) of tasks running successfully (C vs. other languages) . . . 170
210 Performance (min) of tasks running successfully (C# vs. other languages) . . . 171
211 Performance (min) of tasks running successfully (C# vs. other languages) . . . 172
212 Performance (min) of tasks running successfully (F# vs. other languages) . . . 173
213 Performance (min) of tasks running successfully (F# vs. other languages) . . . 174

214 Performance (min) of tasks running successfully (Go vs. other languages) 175
215 Performance (min) of tasks running successfully (Go vs. other languages) 176
216 Performance (min) of tasks running successfully (Haskell vs. other languages) 177
217 Performance (min) of tasks running successfully (Haskell vs. other languages) 178
218 Performance (min) of tasks running successfully (Java vs. other languages) 178
219 Performance (min) of tasks running successfully (Java vs. other languages) 179
220 Performance (min) of tasks running successfully (Python vs. other languages) 179
221 Performance (min) of tasks running successfully (Python vs. other languages) 179
222 Performance (min) of tasks running successfully (all languages) 180
223 Performance (mean) of tasks running successfully (C vs. other languages) 181
224 Performance (mean) of tasks running successfully (C vs. other languages) 182
225 Performance (mean) of tasks running successfully (C# vs. other languages) 183
226 Performance (mean) of tasks running successfully (C# vs. other languages) 184
227 Performance (mean) of tasks running successfully (F# vs. other languages) 185
228 Performance (mean) of tasks running successfully (F# vs. other languages) 186
229 Performance (mean) of tasks running successfully (Go vs. other languages) 187
230 Performance (mean) of tasks running successfully (Go vs. other languages) 188
231 Performance (mean) of tasks running successfully (Haskell vs. other languages) 189
232 Performance (mean) of tasks running successfully (Haskell vs. other languages) 190
233 Performance (mean) of tasks running successfully (Java vs. other languages) 190
234 Performance (mean) of tasks running successfully (Java vs. other languages) 191
235 Performance (mean) of tasks running successfully (Python vs. other languages) 191
236 Performance (mean) of tasks running successfully (Python vs. other languages) 191
237 Performance (mean) of tasks running successfully (all languages) 192
238 Scalability (min) of tasks running successfully (C vs. other languages) 194
239 Scalability (min) of tasks running successfully (C vs. other languages) 195
240 Scalability (min) of tasks running successfully (C# vs. other languages) 196
241 Scalability (min) of tasks running successfully (C# vs. other languages) 197
242 Scalability (min) of tasks running successfully (F# vs. other languages) 198
243 Scalability (min) of tasks running successfully (F# vs. other languages) 199
244 Scalability (min) of tasks running successfully (Go vs. other languages) 200
245 Scalability (min) of tasks running successfully (Go vs. other languages) 201
246 Scalability (min) of tasks running successfully (Haskell vs. other languages) 202
247 Scalability (min) of tasks running successfully (Haskell vs. other languages) 203
248 Scalability (min) of tasks running successfully (Java vs. other languages) 203
249 Scalability (min) of tasks running successfully (Java vs. other languages) 204
250 Scalability (min) of tasks running successfully (Python vs. other languages) 204
251 Scalability (min) of tasks running successfully (Python vs. other languages) 204
252 Scalability (min) of tasks running successfully (all languages) 205
253 Scalability (mean) of tasks running successfully (C vs. other languages) 206

254 Scalability (mean) of tasks running successfully (C vs. other languages) . . . 207
255 Scalability (mean) of tasks running successfully (C# vs. other languages) . . . 208
256 Scalability (mean) of tasks running successfully (C# vs. other languages) . . . 209
257 Scalability (mean) of tasks running successfully (F# vs. other languages) . . . 210
258 Scalability (mean) of tasks running successfully (F# vs. other languages) . . . 211
259 Scalability (mean) of tasks running successfully (Go vs. other languages) . . . 212
260 Scalability (mean) of tasks running successfully (Go vs. other languages) . . . 213
261 Scalability (mean) of tasks running successfully (Haskell vs. other languages) . . . 214
262 Scalability (mean) of tasks running successfully (Haskell vs. other languages) . . . 215
263 Scalability (mean) of tasks running successfully (Java vs. other languages) . . . 215
264 Scalability (mean) of tasks running successfully (Java vs. other languages) . . . 216
265 Scalability (mean) of tasks running successfully (Python vs. other languages) . . . 216
266 Scalability (mean) of tasks running successfully (Python vs. other languages) . . . 216
267 Scalability (mean) of tasks running successfully (all languages) . . . 217
268 Maximum RAM usage (min) of tasks running successfully (C vs. other languages) . . . 219
269 Maximum RAM usage (min) of tasks running successfully (C vs. other languages) . . . 220
270 Maximum RAM usage (min) of tasks running successfully (C# vs. other languages) . . . 221
271 Maximum RAM usage (min) of tasks running successfully (C# vs. other languages) . . . 222
272 Maximum RAM usage (min) of tasks running successfully (F# vs. other languages) . . . 223
273 Maximum RAM usage (min) of tasks running successfully (F# vs. other languages) . . . 224
274 Maximum RAM usage (min) of tasks running successfully (Go vs. other languages) . . . 225
275 Maximum RAM usage (min) of tasks running successfully (Go vs. other languages) . . . 226
276 Maximum RAM usage (min) of tasks running successfully (Haskell vs. other languages) . . . 227
277 Maximum RAM usage (min) of tasks running successfully (Haskell vs. other languages) . . . 228
278 Maximum RAM usage (min) of tasks running successfully (Java vs. other languages) . . . 228
279 Maximum RAM usage (min) of tasks running successfully (Java vs. other languages) . . . 229
280 Maximum RAM usage (min) of tasks running successfully (Python vs. other languages) . . . 229
281 Maximum RAM usage (min) of tasks running successfully (Python vs. other languages) . . . 229
282 Maximum RAM usage (min) of tasks running successfully (all languages) . . . 230
283 Maximum RAM usage (mean) of tasks running successfully (C vs. other languages) . . . 231
284 Maximum RAM usage (mean) of tasks running successfully (C vs. other languages) . . . 232
285 Maximum RAM usage (mean) of tasks running successfully (C# vs. other languages) . . . 233
286 Maximum RAM usage (mean) of tasks running successfully (C# vs. other languages) . . . 234
287 Maximum RAM usage (mean) of tasks running successfully (F# vs. other languages) . . . 235
288 Maximum RAM usage (mean) of tasks running successfully (F# vs. other languages) . . . 236
289 Maximum RAM usage (mean) of tasks running successfully (Go vs. other languages) . . . 237
290 Maximum RAM usage (mean) of tasks running successfully (Go vs. other languages) . . . 238
291 Maximum RAM usage (mean) of tasks running successfully (Haskell vs. other languages) . . . 239
292 Maximum RAM usage (mean) of tasks running successfully (Haskell vs. other languages) . . . 240
293 Maximum RAM usage (mean) of tasks running successfully (Java vs. other languages) . . . 240

294 Maximum RAM usage (mean) of tasks running successfully (Java vs. other languages) 241
295 Maximum RAM usage (mean) of tasks running successfully (Python vs. other languages) 241
296 Maximum RAM usage (mean) of tasks running successfully (Python vs. other languages) 241
297 Maximum RAM usage (mean) of tasks running successfully (all languages) 242
298 Page faults (min) of tasks running successfully (C vs. other languages) 244
299 Page faults (min) of tasks running successfully (C vs. other languages) 245
300 Page faults (min) of tasks running successfully (C# vs. other languages) 246
301 Page faults (min) of tasks running successfully (C# vs. other languages) 247
302 Page faults (min) of tasks running successfully (F# vs. other languages) 248
303 Page faults (min) of tasks running successfully (F# vs. other languages) 249
304 Page faults (min) of tasks running successfully (Go vs. other languages) 250
305 Page faults (min) of tasks running successfully (Go vs. other languages) 251
306 Page faults (min) of tasks running successfully (Haskell vs. other languages) 252
307 Page faults (min) of tasks running successfully (Haskell vs. other languages) 253
308 Page faults (min) of tasks running successfully (Java vs. other languages) 253
309 Page faults (min) of tasks running successfully (Java vs. other languages) 254
310 Page faults (min) of tasks running successfully (Python vs. other languages) 254
311 Page faults (min) of tasks running successfully (Python vs. other languages) 254
312 Page faults (min) of tasks running successfully (all languages) 255
313 Page faults (mean) of tasks running successfully (C vs. other languages) 256
314 Page faults (mean) of tasks running successfully (C vs. other languages) 257
315 Page faults (mean) of tasks running successfully (C# vs. other languages) 258
316 Page faults (mean) of tasks running successfully (C# vs. other languages) 259
317 Page faults (mean) of tasks running successfully (F# vs. other languages) 260
318 Page faults (mean) of tasks running successfully (F# vs. other languages) 261
319 Page faults (mean) of tasks running successfully (Go vs. other languages) 262
320 Page faults (mean) of tasks running successfully (Go vs. other languages) 263
321 Page faults (mean) of tasks running successfully (Haskell vs. other languages) 264
322 Page faults (mean) of tasks running successfully (Haskell vs. other languages) 265
323 Page faults (mean) of tasks running successfully (Java vs. other languages) 265
324 Page faults (mean) of tasks running successfully (Java vs. other languages) 266
325 Page faults (mean) of tasks running successfully (Python vs. other languages) 266
326 Page faults (mean) of tasks running successfully (Python vs. other languages) 266
327 Page faults (mean) of tasks running successfully (all languages) 267
328 Timeout analysis of scalability tasks (C vs. other languages) 269
329 Timeout analysis of scalability tasks (C vs. other languages) 270
330 Timeout analysis of scalability tasks (C# vs. other languages) 271
331 Timeout analysis of scalability tasks (C# vs. other languages) 272
332 Timeout analysis of scalability tasks (F# vs. other languages) 273
333 Timeout analysis of scalability tasks (F# vs. other languages) 274

334 Timeout analysis of scalability tasks (Go vs. other languages) . . . 275
335 Timeout analysis of scalability tasks (Go vs. other languages) . . . 276
336 Timeout analysis of scalability tasks (Haskell vs. other languages) . . . 277
337 Timeout analysis of scalability tasks (Haskell vs. other languages) . . . 278
338 Timeout analysis of scalability tasks (Java vs. other languages) . . . 278
339 Timeout analysis of scalability tasks (Java vs. other languages) . . . 279
340 Timeout analysis of scalability tasks (Python vs. other languages) . . . 279
341 Timeout analysis of scalability tasks (Python vs. other languages) . . . 279
342 Timeout analysis of scalability tasks (all languages) . . . 280
343 Number of solutions per task (C vs. other languages) . . . 282
344 Number of solutions per task (C vs. other languages) . . . 283
345 Number of solutions per task (C# vs. other languages) . . . 284
346 Number of solutions per task (C# vs. other languages) . . . 285
347 Number of solutions per task (F# vs. other languages) . . . 286
348 Number of solutions per task (F# vs. other languages) . . . 287
349 Number of solutions per task (Go vs. other languages) . . . 288
350 Number of solutions per task (Go vs. other languages) . . . 289
351 Number of solutions per task (Haskell vs. other languages) . . . 290
352 Number of solutions per task (Haskell vs. other languages) . . . 291
353 Number of solutions per task (Java vs. other languages) . . . 291
354 Number of solutions per task (Java vs. other languages) . . . 292
355 Number of solutions per task (Python vs. other languages) . . . 292
356 Number of solutions per task (Python vs. other languages) . . . 292
357 Number of solutions per task (all languages) . . . 293

List of Tables

1 Classification and selection of Rosetta Code tasks. . . . 2
2 Rosetta Code ranking: top 20. . . . 3
3 TIOBE index ranking: top 20. . . . 3
4 Combined ranking: the top-2 languages in each category are selected for the study. . . . 3
5 Comparison of lines of code (by minimum). . . . 6
7 Comparison of size of executables (by minimum). . . . 6
9 Computing-intensive tasks. . . . 7
10 Comparison of running time (by minimum) for computing-intensive tasks. . . . 7
12 Comparison of maximum RAM used (by minimum). . . . 8
14 Number of solutions that ran without timeout, and their percentage that ran without errors. . . . 9
15 Comparisons of runtime failure proneness. . . . 9
17 Number of solutions considered for compilation, and their percentage that compiled without errors. . . . 9
18 Names and input size of scalability tasks . . . 27
19 Comparison of conciseness (by min) for tasks compiling successfully . . . 30
22 Comparison of conciseness (by mean) for tasks compiling successfully . . . 32

25 Comparison of conciseness (by min) for all tasks 34

28 Comparison of conciseness (by mean) for all tasks 36

31 Comparison of comments per line of code (by min) for all tasks 38

34 Comparison of comments per line of code (by mean) for all tasks 40

37 Comparison of binary size (by min) for tasks compiling successfully 42

40 Comparison of binary size (by mean) for tasks compiling successfully 44

43 Comparison of performance (by min) for tasks running successfully 46

46 Comparison of performance (by mean) for tasks running successfully 48

49 Comparison of scalability (by min) for tasks running successfully 50

53 Comparison of scalability (by mean) for tasks running successfully 52

57 Comparison of RAM used (by min) for tasks running successfully 54

61 Comparison of RAM used (by mean) for tasks running successfully 56

65 Comparison of page faults (by min) for tasks running successfully 58

68 Comparison of page faults (by mean) for tasks running successfully 60

71 Comparison of time-outs (3 minutes) on scalability tasks 62

74 Comparison of number of solutions per task 64

77 Unpaired comparisons of compilation status 66

79 Unpaired comparisons of running status 67

81 Unpaired comparisons of combined compilation and running 68

83 Unpaired comparisons of runtime fault proneness 69

85 Statistics about the compilation process: columns **make ok** and **make ko** report percentages relative to **solutions** for each language; the columns in between report percentages relative to **make ok** for each language 70

86 Statistics about the running process: all columns *other than* **tasks** and **solutions** report percentages relative to **solutions** for each language 70

87 Statistics about fault proneness: columns **error** and **run ok** report percentages relative to **solutions** for each language 70

For all data processing we used R version 2.14.1. The Wilcoxon signed-rank test and the Mann-Whitney U-test were performed using package `coin` version 1.0-23, except for the test statistics W and U that were computed using R's standard function `wilcox.test`; Cohen's d calculations were performed using package `lsr` version 0.3.2.

VIII. Appendix: Pairwise comparisons

Sections VIII-A to VIII-J describe the complete measured, rendered as graphs and tables, for a number of pairwise comparisons between programming languages; the actual graphs and table appear in the remaining parts of this appendix.

Each comparison targets a different metric M, including lines of code (conciseness), lines of comments per line of code (comments), binary size (in kilobytes, where binaries may be native or byte code), CPU user time (in seconds, for different sets of *performance* T_{PERF}—"everyday" in the main paper—and *scalability* T_{SCAL}—"computing-intensive" in the main paper—tasks), maximum RAM usage (i.e., maximum resident set size, in kilobytes), number of page faults, time outs (with a timeout limit of 3 minutes), and number of Rosetta Code solutions for the same task. Most metrics are *normalized*, as we detail in the subsections.

A metric may also be such that *smaller is better* (such as lines of code: the fewer the more concise a program is) or *larger is better* (such as comments per line of code: the more the more comments are available). Indeed, comments per line of code and number of solutions per task are "larger is better" metrics; all other metrics are "smaller is better". We discuss below how this feature influences how the results should be read.

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.

Using this notation, the comparison of programming 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. T may be further restricted based on a measure-dependent *criterion*, which we describe in the following subsections; for example, Section VIII-A only considers a task t if both X and Y have at least one solution that compiles without errors (solutions that do not satisfy the criterion are discarded). Based on T, we build two data vectors x_M^α and y_M^α for the two languages by aggregating metric M per task using an *aggregation function* α.

To this end, if M is normalized, the normalization factor $\nu_M(t,X,Y)$ denotes the smallest value of M for all solutions of t in X and in Y; otherwise it is just one:

$$\nu_M(t,X,Y) \quad = \quad \begin{cases} \min\left(X_M(t)Y_M(t)\right) & \text{if } M \text{ is normalized and } \min(X_M(t)Y_M(t)) > 0\,, \\ 1 & \text{otherwise}\,, \end{cases}$$

where juxtaposing vectors denotes concatenating them. Note that the normalization factor is one also if M is normalized but the minimum is zero; this is to avoid divisions by zero when normalizing. (A minimum of zero may occur due to the limited precision of some measures such as running time.)

We are finally ready to define vectors x_M^α and y_M^α. The vectors have the same length $|T| = |x_M^\alpha| = |y_M^\alpha|$ and are ordered by task; thus, $x_M^\alpha(t)$ and $y_M^\alpha(t)$ denote the value in x_M^α and in y_M^α corresponding to task t, for $t \in T$:

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

As aggregation functions, we normally consider both minimum and mean; hence the sets of graphs and tables are often double, one for each aggregation function.

The data in x_M^α and y_M^α determines two graphs and a statistical test.

- One graph includes *line plots* of x_M^α and of y_M^α, with the horizontal axis representing task number and the vertical axis representing values of M (possibly normalized).
  For example, Figure 88 includes a graph with normalized values of lines of code aggregated per task by minimum for C and Python. There you can see that there are close to 350 tasks with at least one solution in both C and Python that compiles successfully; and that there is a task whose shortest solution in C is over 50 times larger (in lines of code) than its shortest solution in Python.
- Another graph is a *scatter plot* of x_M^α and of y_M^α, namely of points with coordinates $(x_M^\alpha(t), y_M^\alpha(t))$ for all available tasks $t \in T$. This graph also includes a linear regression line fitted using the least squares approach. Since axes have the same scales in these graphs, a linear regression line that bisects the graph diagonally at 45° would mean that there is no visible difference in metric M between the two languages. Otherwise, if M is such that "smaller is better", the flatter or lower the regression line, the better language Y tends to be compared against language X on metric M. In fact, a flatter or lower line denotes more points (v_X, v_Y) with $v_Y < v_X$ than the other way round, or more tasks where Y is better (smaller metric). Conversely, if M is such that "larger is better", the steeper or higher the regression line, the better language Y tends to be compared against language X on metric M.

For example, Figure 89 includes a graph with normalized values of lines of code aggregated per task by minimum for C and Python. There you can see that most tasks are such that the shortest solution in C is larger than the shortest solution in Python; the regression line is almost horizontal at ordinate 1.

- The *statistical test* is a 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. The test results appear in a table, under column labeled with language X at a row labeled with language Y, and includes various statistics:

1) The p-value is the probability that the differences between x_M^α and y_M^α are due to chance; thus, if p is small (typically at least $p < 0.1$, but preferably $p \ll 0.01$) it means that there is a high chance that X and Y exhibit a genuinely different behavior with respect to metric M. Significant p-values are colored: highly significant ($p < 0.01$) and significant but not highly so ("tends to be significant": $0.01 \leq p < 0.05$).
2) The total sample size N is $|x_M^\alpha| + |x_M^\alpha|$, that is twice the number of tasks considered for metric M.
3) The test statistics W is the absolute value of the sum of the signed ranks (see a description of the test for details).
4) The related test statistics Z is derivable from W.
5) The effect size, computed as Cohen's d, which, for statistically significant differences, gives an idea of how large the difference is. As a rule of thumb, $d < 0.3$ denotes a small effect size, $0.3 \leq d < 0.7$ denotes a medium effect size, and $d \geq 0.7$ denotes a large effect size. Non-negligible effect sizes are colored: large effect size, medium effect size, and small (but non vanishing, that is > 0.05) effect size.
6) The difference $\Delta = \overline{x_M^\alpha} - \overline{y_M^\alpha}$ of the means (which equal the mean difference since the samples have equal size), which gives an unstandardized measure and sign of the size of the difference. Namely, if M is such that "smaller is better" and the difference between X and Y is significant, a positive Δ indicates that language Y is on average better (smaller) on M than language X. Conversely, if M is such that "larger is better", a negative Δ indicates that language Y is on average better (larger) on M than language X.
7) The ratio

$$R \quad = \quad \mathrm{sgn}(\Delta)\,\frac{\max(\overline{x_M^\alpha}, \overline{y_M^\alpha})}{\min(\overline{x_M^\alpha}, \overline{y_M^\alpha})}$$

of the largest mean to the smallest mean, with the same sign as Δ. This is another unstandardized measure and sign of the size of the difference with a more direct interpretation for normalized metrics.

For example, Table 19 includes a cell comparing C (column header) against Python (row header) for normalized values of lines of code aggregated per task by minimum. The p-value is practically zero, and hence the differences are highly significant. The effect size is large ($d > 0.9$), and hence the magnitude of the differences is considerable. Since the metric for conciseness is "smaller is better", a positive Δ indicates that Python is the more concise language on average; the value of R further indicates that the average C solution is over 4.5 times longer in lines of code than the average Python solution. These figures quantitatively confirm what we observed in the line and scatter plots.

We also include a cumulative line plot with all languages at once, which is only meant as a qualitative visualization.

A. Conciseness

The metric for conciseness is non-blank non-comment lines of code, counted using `cloc` version 1.6.2. The metric is normalized and smaller is better. As aggregation functions we consider minimum 'min' and mean. The criterion only selects solutions that compile successfully (compilation returns with exit status 0), and only include tasks T_{LOC} which we manually marked for lines of code count.

B. Conciseness (all tasks)

The metric for conciseness on all tasks is non-blank non-comment lines of code, counted using `cloc` version 1.6.2. The metric is normalized and smaller is better. As aggregation functions we consider minimum 'min' and mean. The criterion only include tasks T_{LOC} which we manually marked for lines of code count (but otherwise includes all solutions, including those that do not compile correctly).

C. Comments

The metric for comments is comment lines of code per non-blank non-comment lines of code, counted using `cloc` version 1.6.2. The metric is normalized and *larger* is better. As aggregation functions we consider minimum 'min' and mean. The criterion only selects tasks T_{LOC} which we manually marked for lines of code count (but otherwise includes all solutions, including those that do not compile correctly).

D. Binary size

The metric for binary size is size of binary (in kilobytes), measured using GNU `du` version 8.13. The metric is normalized and smaller is better. As aggregation functions we consider minimum 'min' and mean. The criterion only selects solutions that compile successfully (compilation returns with exit status 0 and creates a non-empty binary), and only include tasks T_{COMP} which we manually marked for compilation.

The "binary" is either native code or byte code, according to the language. Ruby does not feature in this comparison since it does not generate byte code, and hence the graphs and tables for this metric do not include Ruby.

E. Performance

The metric for performance is CPU user time (in seconds), measured using GNU `time` version 1.7. The metric is normalized and smaller is better. As aggregation functions we consider minimum 'min' and mean. The criterion only selects solutions that execute successfully (execution returns with exit status 0), and only include tasks T_{PERF} which we manually marked for performance comparison.

We selected the performance tasks based on whether they represent well-defined comparable tasks were measuring performance makes sense, and we ascertained that all solutions used in the analysis indeed implement the task correctly (and the solutions are comparable, that is interpret the task consistently and run on comparable inputs).

F. Scalability

The metric for scalability is CPU user time (in seconds), measured using GNU `time` version 1.7. The metric is normalized and smaller is better. As aggregation functions we consider minimum 'min' and mean. The criterion only selects solutions that execute successfully (execution returns with exit status 0), and only include tasks T_{SCAL} which we manually marked for scalability comparison. Table 18 lists the scalability tasks and describes the size n of their inputs in the experiments.

We selected the scalability tasks based on whether they represent well-defined comparable tasks were measuring scalability makes sense. We ascertained that all solutions used in the analysis indeed implement the task correctly (and the solutions are comparable, that is interpret the task consistently); and we modified the input to all solutions so that they are uniform across languages and represent challenging (or at least non-trivial) input sizes.

G. Memory usage

The metric for memory (RAM) usage is maximum resident set size (in kilobytes), measured using GNU `time` version 1.7. The metric is normalized and smaller is better. As aggregation functions we consider minimum 'min' and mean. The criterion only selects solutions that execute successfully (execution returns with exit status 0), and only include tasks T_{SCAL} which we manually marked for scalability comparison.

H. Page faults

The metric for page faults is number of page faults in an execution, measured using GNU `time` version 1.7. The metric is normalized and smaller is better. As aggregation functions we consider minimum 'min' and mean. The criterion only selects solutions that execute successfully (execution returns with exit status 0), and only include tasks T_{SCAL} which we manually marked for scalability comparison.

A number of tests could not be performed due to languages not generating any page faults (all pairs are ties). In those cases, the metric is immaterial.

I. Timeouts

The metric for timeouts is ordinal and two-valued: a solution receives a value of one if it times out within the allotted time; otherwise it receives a value of zero. Time outs were detected using GNU `timeout` version 8.13. The metric is *not* normalized and smaller is better. As aggregation function we consider maximum 'max', corresponding to letting $\ell(t) = 1$ iff all selected solutions to task t in language ℓ time out. The criterion only selects solutions that either execute successfully (execution terminates and returns with exit status 0) or are still running at the timeout limit, and only include tasks T_{SCAL} which we manually marked for scalability comparison.

The line plots for this metric are actually point plots for better readability. Also for readability, the majority of tasks with the same value in the languages under comparison correspond to a different color (marked "all" in the legends).

	TASK NAME	INPUT SIZE n
1	9 billion names of God the integer	$n = 10^5$
2	Anagrams	100 × `unixdict.txt` (20.6 MB)
3	Anagrams/Deranged anagrams	100 × `unixdict.txt` (20.6 MB)
4	Arbitrary-precision integers (included)	$5^{4^{3^2}}$
5	Combinations	$\binom{25}{10}$
6	Count in factors	$n = 10^6$
7	Cut a rectangle	10×10
8	Extensible prime generator	10^7th prime
9	Find largest left truncatable prime in a given base	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	Knapsack problem/0-1	input from Rosetta Code task description
14	Knapsack problem/Bounded	input from Rosetta Code task description
15	Knapsack problem/Continuous	input from Rosetta Code task description
16	Knapsack problem/Unbounded	input from Rosetta Code task description
17	Ludic numbers	input from Rosetta Code task description
18	LZW compression	100 × `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 × `unixdict.txt`
25	Sequence of non-squares	non-squares $< 10^6$
26	Sorting algorithms/Bead sort	$n = 10^4$, nonnegative values $< 10^4$
27	Sorting algorithms/Bubble sort	$n = 3 \cdot 10^4$
28	Sorting algorithms/Cocktail sort	$n = 3 \cdot 10^4$
29	Sorting algorithms/Comb sort	$n = 10^6$
30	Sorting algorithms/Counting sort	$n = 2 \cdot 10^6$, nonnegative values $< 2 \cdot 10^6$
31	Sorting algorithms/Gnome sort	$n = 3 \cdot 10^4$
32	Sorting algorithms/Heapsort	$n = 10^6$
33	Sorting algorithms/Insertion sort	$n = 3 \cdot 10^4$
34	Sorting algorithms/Merge sort	$n = 10^6$
35	Sorting algorithms/Pancake sort	$n = 3 \cdot 10^4$
36	Sorting algorithms/Quicksort	$n = 2 \cdot 10^6$
37	Sorting algorithms/Radix sort	$n = 2 \cdot 10^6$, nonnegative values $< 2 \cdot 10^6$
38	Sorting algorithms/Selection sort	$n = 3 \cdot 10^4$
39	Sorting algorithms/Shell sort	$n = 2 \cdot 10^6$
40	Sorting algorithms/Stooge sort	$n = 3 \cdot 10^3$
41	Sorting algorithms/Strand sort	$n = 3 \cdot 10^4$
42	Text processing/1	input from Rosetta Code task description (1.2 MB)
43	Text processing/2	input from Rosetta Code task description (1.2 MB)
44	Topswops	$n = 12$
45	Towers of Hanoi	$n = 25$
46	Vampire number	input from Rosetta Code task description

Table 18: Names and input size of scalability tasks

J. Solutions per task

The metric for timeouts is a counting metric: each solution receives a value of one. The metric is *not* normalized and *larger* is better. As aggregation function we consider the sum; hence each task receives a value corresponding to the number of solutions in Rosetta Code for that task in each language. The criterion only selects tasks T_{COMP} which we manually marked for compilation (but otherwise includes all solutions, including those that do not compile correctly).

The line plots for this metric are actually point plots for better readability. Also for readability, the majority of tasks with the same value in the languages under comparison correspond to a different color (marked "all" in the legends).

K. Other comparisons

Table 77, Table 79, Table 85, and Table 86 display the results of additional statistics comparing programming languages.

L. Compilation

Table 77 and Table 85 give more details about the compilation process.

Table 77 is similar to the previous tables, but it is based on *unpaired* tests, namely the Mann-Whitney U test—a non-parametric ordinal test that can be applied to two samples of different size. We first consider all solutions to tasks T_{COMP} marked for compilation (regardless of compilation outcome). We then assign an ordinal value to each *solution*:

0: if the solution compiles without errors (the compiler returns with exit status 0 and, if applicable, creates a non-empty binary) with the default compilation options;
1: if the solution compiles without errors (the compiler returns with exit status 0 and, if applicable, creates a non-empty binary), but it requires to set a compilation flag to specify where to find *libraries*;

2: if the solution compiles without errors (the compiler returns with exit status 0 and, if applicable, creates a non-empty binary), but it requires to specify how to *merge* or otherwise process multiple input files;
3: if the solution compiles without errors (the compiler returns with exit status 0 and, if applicable, creates a non-empty binary), but only after applying a *patch*, which deploys some settings (such as include directives);
4: if the solution compiles without errors (the compiler returns with exit status 0 and, if applicable, creates a non-empty binary), but only after *fixing* some simple error (such as a type error, or a missing variable declaration);
5: if the solution does not compile or compiles with errors (the compiler returns with exit status other than 1 or, if applicable, creates no non-empty binary), even after applying possible patches or fixing.

To make the categories disjoint, we assign the highest possible value in each case, reflecting the fact that the lower the ordinal value the better. For example, if a solution requires a patch and a merge, we classify it as a patch, which characterizes the most effort involved in making it compile.

The distinction between patch and fixing is somewhat subjective; it tries to reflect whether the error that had to be rectified was a trivial omission (patch) or a genuine error (fixing). However, we stopped fixing at simple errors, dropping all programs that misinterpreted a task description, referenced obviously missing pieces of code, or required substantial structural modifications to work. All solutions suffering from these problems these received an ordinal value of 5.

For each pair of languages X and Y, a Mann-Whitney U test assessed whether the two samples (ordinal values for language X vs. ordinal values for language Y) come from the same population. The test results appear in Table 77, under column labeled with language X at a row labeled with language Y, and includes various statistics:

1) The p-value is the probability that the two samples come from the same population.
2) The total sample size N is the total number of solutions in language X and Y that received an ordinal value.
3) The test statistics U (see a description of the test for details).
4) The related test statistics Z, derivable from U.
5) The effect size—Cohen's d.
6) The difference Δ of the means, which gives a sign to the difference between the samples. Namely, if p is small, a positive Δ indicates that language Y is on average "better" (fewer compilation problems) than language X.

Table 85 reports, for each language, the number of **tasks** and **solutions** considered for compilation; in column **make ok**, the percentage of solutions that eventually compiled correctly (ordinal values in the range 0–4); in column **make ko**, the percentage of solutions that did not compile correctly (ordinal value 5); in columns **none** through **fix**, the percentage of solutions that eventually compiled correctly for each category corresponding to ordinal values in the range 0–4.

M. Execution

Table 79 and Table 86 give more details about the running process; they are the counterparts to Table 77 and Table 85.

We first consider all solutions to tasks T_{EXEC} marked for execution that we could run. We then assign an ordinal value to each *solution*:

0: if the solution runs without errors (it runs and terminates within the timeout, returns with exit status 0, and does not write anything to standard error);
1: if the solution runs with possible errors but *terminates* successfully (it runs and terminates within the timeout, returns with exit status 0, but writes some messages to standard error);
2: if the solution *times out* (it runs without errors, and it is still running when the time out elapses);
3: if the solution runs with visible *error* (it runs and terminates within the timeout, returns with exit status other than 0, and writes some messages to standard error);
4: if the solution *crashes* (it runs and terminates within the timeout, returns with exit status other than 0, and writes nothing to standard error, such as a `Segfault`).

The categories are disjoint and try to reflect increasing levels of problems. We consider terminating without printing error (a crash) worse than printing some information. Similarly, we consider nontermination without manifest error better than abrupt termination with error. (In fact, many solutions in this categories were either from correct solutions working on very large inputs, typically in the scalability tasks, or to correct solutions to *interactive* tasks were termination is not to be expected.) The distinctions are somewhat subjective; they try to reflect the difficulty of understanding and possibly debugging an error.

For each pair of languages X and Y, a Mann-Whitney U test assessed whether the two samples (ordinal values for language X vs. ordinal values for language Y) come from the same population. The test results appear in Table 79, under column labeled with language X at a row labeled with language Y, and includes the same statistics as Table 77.

Table 86 reports, for each language, the number of **tasks** and **solutions** considered for execution; in columns **run ok** through **crash**, the percentage of solutions for each category corresponding to ordinal values in the range 0–4.

N. Overall code quality (compilation + execution)

Table 81 compares the *sum* of ordinal values assigned to each *solution* as described in Section VIII-L and Section VIII-M, for all tasks T_{EXEC} marked for execution (and compilation). The overall score gives an idea of the code quality of Rosetta Code solutions based on how much we had to do for compilation and for execution.

O. Fault proneness

Table 83 and Table 87 give an idea of the number of defects manifesting themselves as runtime failures; they draw on data similar to those presented in Section VIII-L and Section VIII-M.

We first consider all solutions to tasks T_{EXEC} marked for execution that we could compile without errors and that ran without timing out. We then assign an ordinal value to each *solution*:

0: if the solution runs without errors (it runs and terminates within the timeout with exit status 0);
1: if the solution runs with errors (it runs and terminates within the timeout with exit status other than 0).

The categories are disjoint and do not include solutions that timed out.

For each pair of languages X and Y, a Mann-Whitney U test assessed whether the two samples (ordinal values for language X vs. ordinal values for language Y) come from the same population. The test results appear in Table 83, under column labeled with language X at a row labeled with language Y, and includes the same statistics as Table 77.

Table 87 reports, for each language, the number of **tasks** and **solutions** that compiled correctly and ran without timing out; in columns **error** and **run ok**, the percentage of solutions for each category corresponding to ordinal values 1 (**error**) and 0 (**run ok**).

Visualizations of language comparisons

To help visualize the results, a graph accompanies each table with the results of statistical significance tests between pairs of languages. 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 how ℓ_1 and ℓ_2 compare according to the statistical test. In contrast, vertical distances are chosen only to improve readability and carry not meaning.

Precisely, there are two graphs for each measure: unnormalized and normalized. In the unnormalized graph, let $p_M(\ell_1,\ell_2)$, $e_M(\ell_1,\ell_2)$, and $\Delta_M(\ell_1,\ell_2)$ be the p-value, effect size (d or r according to the metric), and difference of means for the test that compares ℓ_1 to ℓ_2 on metric M. If $p_M(\ell_1,\ell_2) > 0.05$ or $e_M(\ell_1,\ell_2) < 0.05$, then the horizontal distance between node ℓ_1 and node ℓ_2 is zero or very small (it may still be non-zero to improve the visual layout while satisfying the constraints as well as possible), and there is no edge between them. Otherwise, their horizontal distance is proportional to $\sqrt{e_M(\ell_1,\ell_2)\cdot(1-p_M(\ell_1,\ell_2))}$, and there is a directed edge to the node corresponding to the "better" language according to M from the other node. To improve the visual layout, edges that express an ordered pair that is subsumed by others are omitted, that is if $\ell_a \to \ell_b \to \ell_c$ the edge from ℓ_a to ℓ_c is omitted. Paths on the graph following edges in the direction of the arrow list languages in approximate order from "worse" to "better". Edges are dotted if they correspond to small significant p-values ($0.01 \le p < 0.05$); they have a color corresponding to the color assigned to the effect size in the table: large effect size, medium effect size, and small but non vanishing effect size.

In the *normalized* companion graphs (called with "normalized horizontal distances" in the captions), arrows and vertical distances are assigned as in the unnormalized graphs, whereas the horizontal distance are proportional to the mean $\overline{\ell_M^{\alpha}}$ for each language ℓ over all common tasks (that is where each language has at least one solution). Namely, the language with the "worst" average measure (consistent with whether M is such that "smaller" or "larger" is better) will be farthest on the left, the language with the "best" average measure will be farthest on the right, with other languages in between proportionally to their rank. Since, to have a unique baseline, normalized average measures only refer to tasks common to all languages, the normalized horizontal distances may be inconsistent with the pairwise tests because they refer to a much smaller set of values (sensitive to noise). This is only visible in the case of performance and scalability tasks, which are often sufficiently many for pairwise comparisons, but become too few (for example, less than 10) when we only look at the tasks that have implementations in all languages. In these cases, the unnormalized graphs may be more indicative (and, in any case, the data in the tables is the hard one).

For comparisons based on ordinal values, there is only one kind of graph whose horizontal distances do not have a significant quantitative meaning but mainly represent an ordering.

Remind that all graphs use approximations and heuristics to build their layout; hence they are mainly meant as qualitative visualization aids that cannot substitute a detailed analytical reading of the data.

IX. Appendix: Tables and graphs

A. Lines of code (tasks compiling successfully)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	5.433E-01						
	N	4.640E+02						
	W	1.245E+04						
	Z	-6.078E-01						
	d	3.918E-03						
	Δ	3.112E-03						
	R	1.002E+00						
F#	p	0.000E+00	0.000E+00					
	N	3.640E+02	3.460E+02					
	W	1.542E+04	1.469E+04					
	Z	1.083E+01	1.109E+01					
	d	7.353E-01	9.453E-01					
	Δ	4.090E+00	3.190E+00					
	R	4.919E+00	4.127E+00					
Go	p	3.774E-01	8.210E-02	3.693E-30				
	N	6.340E+02	4.740E+02	3.800E+02				
	W	2.501E+04	1.416E+04	3.320E+02				
	Z	8.827E-01	1.739E+00	-1.141E+01				
	d	1.554E-01	8.305E-02	6.396E-01				
	Δ	8.791E-02	7.200E-02	-3.586E+00				
	R	1.069E+00	1.053E+00	-4.497E+00				
Haskell	p	0.000E+00	0.000E+00	1.675E-01	0.000E+00			
	N	5.540E+02	4.320E+02	3.540E+02	5.600E+02			
	W	3.695E+04	2.275E+04	7.878E+03	3.835E+04			
	Z	1.398E+01	1.241E+01	1.380E+00	1.431E+01			
	d	1.071E+00	1.286E+00	8.453E-02	1.255E+00			
	Δ	2.821E+00	2.742E+00	9.601E-02	2.527E+00			
	R	3.768E+00	3.693E+00	1.064E+00	3.509E+00			
Java	p	2.645E-02	4.328E-05	6.839E-26	2.644E-02	1.433E-33		
	N	4.940E+02	3.900E+02	3.140E+02	4.960E+02	4.400E+02		
	W	1.618E+04	1.194E+04	1.345E+02	1.631E+04	5.830E+02		
	Z	2.219E+00	4.089E+00	-1.052E+01	2.220E+00	-1.207E+01		
	d	2.615E-01	3.186E-01	7.530E-01	1.475E-01	1.052E+00		
	Δ	1.881E-01	2.235E-01	-2.604E+00	9.180E-02	-2.519E+00		
	R	1.150E+00	1.182E+00	-3.566E+00	1.073E+00	-3.416E+00		
Python	p	0.000E+00	0.000E+00	1.132E-05	0.000E+00	2.123E-02	0.000E+00	
	N	6.860E+02	5.080E+02	4.060E+02	7.000E+02	6.060E+02	5.440E+02	
	W	5.650E+04	3.172E+04	1.216E+04	5.826E+04	2.264E+04	3.559E+04	
	Z	1.533E+01	1.362E+01	4.390E+00	1.549E+01	2.304E+00	1.365E+01	
	d	9.511E-01	1.114E+00	3.588E-01	8.157E-01	2.085E-01	9.381E-01	
	Δ	3.666E+00	3.882E+00	4.770E-01	3.633E+00	2.322E-01	3.000E+00	
	R	4.519E+00	4.837E+00	1.329E+00	4.482E+00	1.159E+00	3.886E+00	
Ruby	p	0.000E+00	0.000E+00	1.342E-02	0.000E+00	7.643E-01	0.000E+00	1.522E-02
	N	6.560E+02	5.040E+02	4.000E+02	6.760E+02	5.900E+02	5.400E+02	7.500E+02
	W	5.109E+04	3.045E+04	1.041E+04	5.574E+04	1.913E+04	3.421E+04	2.504E+04
	Z	1.484E+01	1.309E+01	2.472E+00	1.539E+01	-2.999E-01	1.347E+01	-2.427E+00
	d	5.577E-01	8.824E-01	1.026E-01	7.419E-01	1.069E-01	7.625E-01	2.028E-02
	Δ	4.294E+00	3.912E+00	2.409E-01	3.657E+00	1.801E-01	3.018E+00	8.089E-02
	R	5.156E+00	4.758E+00	1.143E+00	4.579E+00	1.114E+00	3.927E+00	1.046E+00

Table 19: Comparison of conciseness (by min) for tasks compiling successfully

Figure 20: Lines of code (min) of tasks compiling successfully

Figure 21: Lines of code (min) of tasks compiling successfully (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	5.898E-01						
	N	4.640E+02						
	W	1.359E+04						
	Z	5.391E-01						
	d	1.146E-01						
	Δ	1.009E-01						
	R	1.068E+00						
F#	p	0.000E+00	0.000E+00					
	N	3.640E+02	3.460E+02					
	W	1.617E+04	1.470E+04					
	Z	1.124E+01	1.111E+01					
	d	7.245E-01	9.136E-01					
	Δ	4.529E+00	3.370E+00					
	R	5.002E+00	4.026E+00					
Go	p	9.736E-02	2.678E-01	1.140E-30				
	N	6.340E+02	4.740E+02	3.800E+02				
	W	2.689E+04	1.416E+04	2.565E+02				
	Z	1.658E+00	1.108E+00	-1.151E+01				
	d	1.008E-01	4.938E-02	6.507E-01				
	Δ	8.861E-02	4.276E-02	-3.746E+00				
	R	1.063E+00	1.030E+00	-4.368E+00				
Haskell	p	0.000E+00	0.000E+00	3.940E-01	0.000E+00			
	N	5.540E+02	4.320E+02	3.540E+02	5.600E+02			
	W	3.728E+04	2.277E+04	7.726E+03	3.892E+04			
	Z	1.405E+01	1.243E+01	8.524E-01	1.438E+01			
	d	9.920E-01	1.266E+00	2.400E-02	1.252E+00			
	Δ	3.050E+00	2.781E+00	-3.257E-02	2.557E+00			
	R	3.623E+00	3.426E+00	-1.019E+00	3.249E+00			
Java	p	3.469E-03	2.188E-04	2.836E-26	3.143E-02	3.333E-34		
	N	4.940E+02	3.900E+02	3.140E+02	4.960E+02	4.400E+02		
	W	1.775E+04	1.188E+04	1.090E+02	1.732E+04	5.350E+02		
	Z	2.923E+00	3.696E+00	-1.060E+01	2.152E+00	-1.219E+01		
	d	2.972E-01	1.988E-01	5.150E-01	1.594E-01	1.020E+00		
	Δ	2.336E-01	1.690E-01	-3.069E+00	1.035E-01	-2.540E+00		
	R	1.176E+00	1.127E+00	-3.915E+00	1.078E+00	-3.125E+00		
Python	p	0.000E+00	0.000E+00	1.067E-01	0.000E+00	6.908E-01	0.000E+00	
	N	6.860E+02	5.080E+02	4.060E+02	7.000E+02	6.060E+02	5.440E+02	
	W	5.647E+04	3.128E+04	1.083E+04	5.837E+04	2.081E+04	3.593E+04	
	Z	1.515E+01	1.325E+01	1.613E+00	1.539E+01	-3.978E-01	1.337E+01	
	d	8.660E-01	9.653E-01	2.510E-02	7.540E-01	7.594E-02	6.005E-01	
	Δ	3.671E+00	3.584E+00	-4.721E-02	3.574E+00	-1.336E-01	2.967E+00	
	R	3.574E+00	3.380E+00	-1.023E+00	3.473E+00	-1.071E+00	3.081E+00	
Ruby	p	0.000E+00	0.000E+00	6.440E-01	0.000E+00	1.344E-02	0.000E+00	2.928E-01
	N	6.560E+02	5.040E+02	4.000E+02	6.760E+02	5.900E+02	5.400E+02	7.500E+02
	W	5.063E+04	3.038E+04	8.814E+03	5.612E+04	1.607E+04	3.404E+04	3.007E+04
	Z	1.457E+01	1.302E+01	-4.621E-01	1.545E+01	-2.472E+00	1.297E+01	-1.052E+00
	d	5.421E-01	7.861E-01	9.270E-02	6.992E-01	3.851E-02	6.834E-01	3.108E-02
	Δ	4.505E+00	3.651E+00	-2.464E-01	3.579E+00	-7.610E-02	2.842E+00	1.294E-01
	R	4.256E+00	3.459E+00	-1.120E+00	3.605E+00	-1.039E+00	3.025E+00	1.059E+00

Table 22: Comparison of conciseness (by mean) for tasks compiling successfully

Figure 23: Lines of code (mean) of tasks compiling successfully

Figure 24: Lines of code (mean) of tasks compiling successfully (normalized horizontal distances)

B. Lines of code (all tasks)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	4.401E-01						
	N	5.460E+02						
	W	1.704E+04						
	Z	-7.720E-01						
	d	1.075E-01						
	Δ	2.202E-01						
	R	1.145E+00						
F#	p	0.000E+00	0.000E+00					
	N	4.160E+02	3.760E+02					
	W	1.948E+04	1.669E+04					
	Z	1.110E+01	1.131E+01					
	d	6.994E-01	9.145E-01					
	Δ	3.552E+00	3.066E+00					
	R	4.304E+00	3.980E+00					
Go	p	9.913E-01	2.754E-01	1.161E-31				
	N	7.360E+02	5.440E+02	4.220E+02				
	W	3.144E+04	1.774E+04	6.315E+02				
	Z	1.090E-02	1.091E+00	-1.171E+01				
	d	3.998E-02	1.160E-01	6.003E-01				
	Δ	-1.049E-01	-3.740E-01	-3.659E+00				
	R	-1.072E+00	-1.253E+00	-4.512E+00				
Haskell	p	0.000E+00	0.000E+00	2.036E-01	0.000E+00			
	N	6.840E+02	5.260E+02	4.120E+02	6.780E+02			
	W	5.371E+04	3.146E+04	1.024E+04	5.348E+04			
	Z	1.473E+01	1.266E+01	1.271E+00	1.460E+01			
	d	7.998E-01	8.051E-01	1.611E-01	7.639E-01			
	Δ	3.183E+00	2.929E+00	3.303E-01	2.892E+00			
	R	4.002E+00	3.510E+00	1.212E+00	3.669E+00			
Java	p	1.298E-03	9.403E-06	1.247E-26	3.332E-04	4.786E-41		
	N	6.500E+02	5.220E+02	4.040E+02	6.440E+02	6.200E+02		
	W	2.936E+04	2.107E+04	1.160E+03	2.868E+04	2.399E+03		
	Z	3.216E+00	4.430E+00	-1.068E+01	3.588E+00	-1.342E+01		
	d	1.419E-01	6.627E-02	5.330E-01	1.513E-01	9.305E-01		
	Δ	2.417E-01	1.335E-01	-2.724E+00	2.126E-01	-2.471E+00		
	R	1.165E+00	1.091E+00	-3.417E+00	1.152E+00	-3.279E+00		
Python	p	0.000E+00	0.000E+00	2.631E-05	0.000E+00	6.806E-02	0.000E+00	
	N	7.560E+02	5.520E+02	4.300E+02	7.540E+02	6.960E+02	6.700E+02	
	W	6.570E+04	3.603E+04	1.324E+04	6.559E+04	2.870E+04	5.211E+04	
	Z	1.555E+01	1.345E+01	4.203E+00	1.519E+01	1.825E+00	1.462E+01	
	d	8.906E-01	9.780E-01	1.913E-01	7.315E-01	5.456E-02	9.323E-01	
	Δ	3.421E+00	3.505E+00	3.405E-01	3.531E+00	8.176E-02	2.793E+00	
	R	4.211E+00	4.108E+00	1.218E+00	4.201E+00	1.050E+00	3.610E+00	
Ruby	p	0.000E+00	0.000E+00	2.048E-02	0.000E+00	5.191E-01	0.000E+00	1.706E-02
	N	7.220E+02	5.500E+02	4.200E+02	7.180E+02	6.700E+02	6.620E+02	7.580E+02
	W	5.924E+04	3.479E+04	1.116E+04	6.057E+04	2.400E+04	4.890E+04	2.557E+04
	Z	1.491E+01	1.284E+01	2.317E+00	1.513E+01	-6.448E-01	1.402E+01	-2.385E+00
	d	5.411E-01	7.852E-01	3.689E-02	6.791E-01	2.995E-02	8.619E-01	1.925E-02
	Δ	4.005E+00	3.638E+00	9.957E-02	3.611E+00	5.625E-02	2.607E+00	7.641E-02
	R	4.731E+00	4.269E+00	1.055E+00	4.311E+00	1.033E+00	3.429E+00	1.043E+00

Table 25: Comparison of conciseness (by min) for all tasks

Figure 26: Lines of code (min) of all tasks

Figure 27: Lines of code (min) of all tasks (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	2.617E-01						
	N	5.460E+02						
	W	1.959E+04						
	Z	1.122E+00						
	d	6.772E-02						
	Δ	3.026E-01						
	R	1.158E+00						
F#	p	0.000E+00	0.000E+00					
	N	4.160E+02	3.760E+02					
	W	2.077E+04	1.739E+04					
	Z	1.180E+01	1.160E+01					
	d	7.332E-01	8.851E-01					
	Δ	4.326E+00	3.274E+00					
	R	4.743E+00	3.790E+00					
Go	p	2.808E-01	8.728E-01	6.487E-33				
	N	7.360E+02	5.440E+02	4.220E+02				
	W	3.481E+04	1.742E+04	4.530E+02				
	Z	1.079E+00	1.601E-01	-1.195E+01				
	d	2.530E-02	4.081E-02	6.160E-01				
	Δ	-7.166E-02	-1.775E-01	-3.873E+00				
	R	-1.042E+00	-1.097E+00	-4.305E+00				
Haskell	p	0.000E+00	0.000E+00	5.726E-01	0.000E+00			
	N	6.840E+02	5.260E+02	4.120E+02	6.780E+02			
	W	5.551E+04	3.281E+04	1.031E+04	5.484E+04			
	Z	1.510E+01	1.306E+01	5.643E-01	1.489E+01			
	d	7.255E-01	6.478E-01	6.647E-02	6.996E-01			
	Δ	3.573E+00	3.284E+00	1.534E-01	3.001E+00			
	R	3.632E+00	3.333E+00	1.082E+00	3.283E+00			
Java	p	3.929E-06	2.200E-06	4.297E-28	6.609E-05	3.959E-41		
	N	6.500E+02	5.220E+02	4.040E+02	6.440E+02	6.200E+02		
	W	3.254E+04	2.174E+04	1.011E+03	3.115E+04	2.629E+03		
	Z	4.615E+00	4.734E+00	-1.099E+01	3.990E+00	-1.343E+01		
	d	2.289E-01	1.069E-01	4.663E-01	1.699E-01	7.125E-01		
	Δ	4.379E-01	3.810E-01	-3.119E+00	2.729E-01	-2.475E+00		
	R	1.275E+00	1.235E+00	-3.464E+00	1.180E+00	-2.757E+00		
Python	p	0.000E+00	0.000E+00	2.070E-01	0.000E+00	8.144E-01	0.000E+00	
	N	7.560E+02	5.520E+02	4.300E+02	7.540E+02	6.960E+02	6.700E+02	
	W	6.703E+04	3.578E+04	1.174E+04	6.625E+04	2.822E+04	5.274E+04	
	Z	1.543E+01	1.326E+01	1.262E+00	1.521E+01	-2.347E-01	1.418E+01	
	d	8.194E-01	7.497E-01	7.523E-02	6.907E-01	6.446E-02	5.965E-01	
	Δ	3.612E+00	3.491E+00	-1.719E-01	3.482E+00	-1.521E-01	2.724E+00	
	R	3.420E+00	3.128E+00	-1.084E+00	3.279E+00	-1.074E+00	2.769E+00	
Ruby	p	0.000E+00	0.000E+00	4.509E-01	0.000E+00	3.112E-02	0.000E+00	3.708E-01
	N	7.220E+02	5.500E+02	4.200E+02	7.180E+02	6.700E+02	6.620E+02	7.580E+02
	W	6.003E+04	3.544E+04	9.528E+03	6.219E+04	2.197E+04	4.965E+04	3.107E+04
	Z	1.485E+01	1.300E+01	-7.539E-01	1.549E+01	-2.156E+00	1.366E+01	-8.949E-01
	d	5.314E-01	7.255E-01	1.101E-01	6.592E-01	1.077E-03	7.406E-01	3.661E-02
	Δ	4.491E+00	3.506E+00	-3.285E-01	3.591E+00	3.226E-03	2.549E+00	1.520E-01
	R	4.149E+00	3.286E+00	-1.157E+00	3.511E+00	1.001E+00	2.759E+00	1.069E+00

Table 28: Comparison of conciseness (by mean) for all tasks

Figure 29: Lines of code (mean) of all tasks

Figure 30: Lines of code (mean) of all tasks (normalized horizontal distances)

C. Comments per line of code

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	1.280E-02						
	N	5.460E+02						
	W	5.485E+03						
	Z	2.489E+00						
	d	1.196E-01						
	Δ	1.312E-01						
	R	1.622E+00						
F#	p	2.582E-03	1.438E-03					
	N	4.160E+02	3.760E+02					
	W	1.762E+03	8.215E+02					
	Z	-3.014E+00	-3.187E+00					
	d	2.267E-01	2.553E-01					
	Δ	-1.245E+00	-4.638E-01					
	R	-7.680E+00	-3.903E+00					
Go	p	2.708E-04	1.590E-04	1.787E-02				
	N	7.360E+02	5.440E+02	4.220E+02				
	W	8.032E+03	2.822E+03	3.651E+03				
	Z	-3.642E+00	-3.777E+00	2.368E+00				
	d	1.483E-01	8.830E-02	2.302E-01				
	Δ	-2.507E-01	-1.023E-01	7.585E-01				
	R	-1.718E+00	-1.449E+00	4.956E+00				
Haskell	p	8.026E-01	9.087E-02	2.919E-03	2.006E-03			
	N	6.840E+02	5.260E+02	4.120E+02	6.780E+02			
	W	8.228E+03	2.120E+03	2.292E+03	9.460E+03			
	Z	2.499E-01	-1.691E+00	2.976E+00	3.089E+00			
	d	1.156E-01	1.839E-01	1.080E-01	2.662E-02			
	Δ	-1.293E-01	-3.060E-01	4.895E-01	-2.255E-02			
	R	-1.546E+00	-3.487E+00	3.560E+00	-1.088E+00			
Java	p	3.675E-03	5.161E-01	4.373E-05	7.217E-06	8.632E-03		
	N	6.500E+02	5.220E+02	4.040E+02	6.440E+02	6.200E+02		
	W	8.242E+03	2.432E+03	2.586E+03	9.179E+03	4.854E+03		
	Z	2.905E+00	-6.493E-01	4.087E+00	4.487E+00	2.626E+00		
	d	3.301E-02	5.305E-02	2.337E-01	1.655E-01	1.544E-01		
	Δ	4.071E-02	1.031E-01	6.728E-01	4.832E-01	4.959E-01		
	R	1.163E+00	1.348E+00	8.473E+00	3.279E+00	5.153E+00		
Python	p	8.629E-01	1.313E-01	2.052E-04	2.637E-03	1.439E-01	1.072E-01	
	N	7.560E+02	5.520E+02	4.300E+02	7.540E+02	6.960E+02	6.700E+02	
	W	8.917E+03	2.174E+03	2.394E+03	1.136E+04	5.716E+03	3.394E+03	
	Z	1.727E-01	-1.509E+00	3.713E+00	3.007E+00	1.461E+00	-1.611E+00	
	d	1.303E-01	9.636E-02	2.101E-01	2.011E-02	5.100E-02	1.486E-01	
	Δ	-1.430E-01	-1.124E-01	4.456E-01	2.140E-02	8.801E-02	-2.691E-01	
	R	-1.693E+00	-1.591E+00	3.321E+00	1.078E+00	1.357E+00	-2.995E+00	
Ruby	p	2.721E-02	2.457E-03	5.496E-03	8.325E-01	3.642E-01	3.449E-03	5.222E-02
	N	7.220E+02	5.500E+02	4.200E+02	7.180E+02	6.700E+02	6.620E+02	7.580E+02
	W	7.644E+03	2.338E+03	2.738E+03	8.540E+03	5.039E+03	3.474E+03	5.300E+03
	Z	-2.208E+00	-3.029E+00	2.776E+00	-2.115E-01	-9.074E-01	-2.925E+00	-1.941E+00
	d	1.983E-01	1.441E-01	2.170E-01	5.172E-02	5.629E-02	1.178E-01	1.088E-01
	Δ	-2.708E-01	-3.446E-01	6.882E-01	-7.159E-02	6.601E-02	-5.201E-01	-1.397E-01
	R	-2.463E+00	-2.670E+00	3.973E+00	-1.242E+00	1.343E+00	-3.497E+00	-1.737E+00

Table 31: Comparison of comments per line of code (by min) for all tasks

Figure 32: Comments per line of code (min) of all tasks

Figure 33: Comments per line of code (min) of all tasks (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	1.126E-04						
	N	5.460E+02						
	W	9.117E+03						
	Z	3.862E+00						
	d	1.451E-01						
	Δ	1.740E-01						
	R	1.744E+00						
F#	p	2.433E-02	4.213E-04					
	N	4.160E+02	3.760E+02					
	W	2.972E+03	1.110E+03					
	Z	-2.252E+00	-3.526E+00					
	d	2.223E-01	2.515E-01					
	Δ	-1.225E+00	-4.586E-01					
	R	-6.234E+00	-3.629E+00					
Go	p	4.530E-03	2.090E-05	4.463E-02				
	N	7.360E+02	5.440E+02	4.220E+02				
	W	1.233E+04	3.448E+03	4.752E+03				
	Z	-2.839E+00	-4.255E+00	2.008E+00				
	d	1.445E-01	1.226E-01	2.258E-01				
	Δ	-2.597E-01	-1.528E-01	7.448E-01				
	R	-1.663E+00	-1.663E+00	4.480E+00				
Haskell	p	9.046E-01	1.451E-02	1.597E-02	2.478E-03			
	N	6.840E+02	5.260E+02	4.120E+02	6.780E+02			
	W	1.294E+04	2.746E+03	3.473E+03	1.338E+04			
	Z	1.199E-01	-2.444E+00	2.410E+00	3.026E+00			
	d	1.196E-01	1.826E-01	1.095E-01	3.122E-02			
	Δ	-1.553E-01	-3.056E-01	4.969E-01	-2.770E-02			
	R	-1.583E+00	-3.229E+00	3.417E+00	-1.100E+00			
Java	p	5.115E-04	9.759E-02	6.576E-04	3.450E-05	1.930E-03		
	N	6.500E+02	5.220E+02	4.040E+02	6.440E+02	6.200E+02		
	W	1.254E+04	3.376E+03	3.647E+03	1.258E+04	8.654E+03		
	Z	3.475E+00	-1.657E+00	3.407E+00	4.141E+00	3.101E+00		
	d	5.711E-02	4.856E-02	2.280E-01	1.813E-01	1.538E-01		
	Δ	7.417E-02	1.013E-01	6.582E-01	5.410E-01	4.944E-01		
	R	1.272E+00	1.288E+00	6.624E+00	3.352E+00	4.581E+00		
Python	p	2.615E-01	5.322E-04	7.370E-03	1.538E-01	7.819E-01	1.694E-03	
	N	7.560E+02	5.520E+02	4.300E+02	7.540E+02	6.960E+02	6.700E+02	
	W	1.348E+04	3.376E+03	3.951E+03	1.497E+04	1.058E+04	6.313E+03	
	Z	-1.123E+00	-3.464E+00	2.680E+00	1.426E+00	2.768E-01	-3.139E+00	
	d	1.452E-01	1.291E-01	1.822E-01	2.651E-02	2.543E-02	1.609E-01	
	Δ	-2.248E-01	-1.602E-01	3.940E-01	-3.184E-02	4.544E-02	-2.997E-01	
	R	-1.887E+00	-1.794E+00	2.563E+00	-1.100E+00	1.150E+00	-2.925E+00	
Ruby	p	1.955E-02	5.776E-05	2.830E-02	6.545E-01	2.017E-01	1.513E-03	1.701E-01
	N	7.220E+02	5.500E+02	4.200E+02	7.180E+02	6.700E+02	6.620E+02	7.580E+02
	W	1.205E+04	3.519E+03	3.986E+03	1.217E+04	8.104E+03	5.571E+03	1.096E+04
	Z	-2.335E+00	-4.022E+00	2.193E+00	-4.475E-01	-1.277E+00	-3.172E+00	-1.372E+00
	d	1.663E-01	1.566E-01	2.103E-01	3.913E-02	5.035E-02	1.196E-01	8.777E-02
	Δ	-3.391E-01	-3.784E-01	6.673E-01	-5.510E-02	5.931E-02	-5.300E-01	-1.201E-01
	R	-2.440E+00	-2.723E+00	3.557E+00	-1.166E+00	1.277E+00	-3.193E+00	-1.493E+00

Table 34: Comparison of comments per line of code (by mean) for all tasks

Figure 35: Comments per line of code (mean) of all tasks

Figure 36: Comments per line of code (mean) of all tasks (normalized horizontal distances)

D. Size of binaries

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java
C#	p	0.000E+00					
	N	4.660E+02					
	W	2.611E+04					
	Z	1.349E+01					
	d	2.669E+00					
	Δ	1.411E+00					
	R	2.411E+00					
F#	p	0.000E+00	3.829E-16				
	N	3.620E+02	3.460E+02				
	W	1.239E+04	5.265E+02				
	Z	1.017E+01	-8.144E+00				
	d	1.395E+00	1.267E+00				
	Δ	6.625E-01	-5.800E-01				
	R	1.639E+00	-1.551E+00				
Go	p	1.802E-53	1.705E-40	1.270E-32			
	N	6.300E+02	4.720E+02	3.760E+02			
	W	0.000E+00	0.000E+00	0.000E+00			
	Z	-1.539E+01	-1.332E+01	-1.189E+01			
	d	3.639E+00	2.312E+00	2.403E+00			
	Δ	-1.533E+02	-3.860E+02	-2.569E+02			
	R	-1.543E+02	-3.870E+02	-2.579E+02			
Haskell	p	6.818E-46	3.241E-36	5.508E-30	0.000E+00		
	N	5.380E+02	4.200E+02	3.440E+02	5.380E+02		
	W	0.000E+00	0.000E+00	0.000E+00	3.037E+04		
	Z	-1.422E+01	-1.257E+01	-1.138E+01	9.562E+00		
	d	2.469E+00	2.224E+00	2.544E+00	1.071E+00		
	Δ	-1.094E+02	-2.663E+02	-1.726E+02	4.267E-01		
	R	-1.104E+02	-2.673E+02	-1.736E+02	1.378E+00		
Java	p	0.000E+00	3.330E-05	0.000E+00	0.000E+00	0.000E+00	
	N	4.920E+02	3.900E+02	3.140E+02	4.920E+02	4.280E+02	
	W	2.788E+04	2.484E+03	7.784E+03	3.038E+04	2.300E+04	
	Z	1.377E+01	4.150E+00	9.691E+00	1.360E+01	1.269E+01	
	d	3.148E+00	3.638E-01	1.680E+00	3.121E+00	1.591E+00	
	Δ	1.755E+00	1.761E-01	8.502E-01	4.136E+02	3.121E+02	
	R	2.706E+00	1.156E+00	1.821E+00	4.146E+02	3.131E+02	
Python	p	0.000E+00	2.220E-16	0.000E+00	0.000E+00	0.000E+00	2.067E-06
	N	6.860E+02	5.100E+02	4.060E+02	6.960E+02	5.880E+02	5.460E+02
	W	5.900E+04	5.686E+03	1.240E+04	6.073E+04	4.336E+04	2.076E+03
	Z	1.722E+01	8.233E+00	1.151E+01	1.618E+01	1.487E+01	4.747E+00
	d	5.686E+00	8.992E-01	1.517E+00	3.430E+00	1.676E+00	3.945E-01
	Δ	2.028E+00	3.686E-01	1.069E+00	4.747E+02	3.519E+02	3.077E-01
	R	3.028E+00	1.352E+00	2.069E+00	4.757E+02	3.529E+02	1.287E+00

Table 37: Comparison of binary size (by min) for tasks compiling successfully

Figure 38: Size of binaries (min) of tasks compiling successfully

Figure 39: Size of binaries (min) of tasks compiling successfully (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java
C#	p	0.000E+00					
	N	4.660E+02					
	W	2.634E+04					
	Z	1.343E+01					
	d	2.633E+00					
	Δ	1.409E+00					
	R	2.386E+00					
F#	p	0.000E+00	1.234E-16				
	N	3.620E+02	3.460E+02				
	W	1.270E+04	5.290E+02				
	Z	1.014E+01	-8.280E+00				
	d	1.366E+00	1.267E+00				
	Δ	6.519E-01	-5.844E-01				
	R	1.616E+00	-1.544E+00				
Go	p	1.948E-53	1.765E-40	1.301E-32			
	N	6.300E+02	4.720E+02	3.760E+02			
	W	0.000E+00	0.000E+00	0.000E+00			
	Z	-1.539E+01	-1.332E+01	-1.189E+01			
	d	3.662E+00	2.327E+00	2.413E+00			
	Δ	-1.540E+02	-3.891E+02	-2.595E+02			
	R	-1.539E+02	-3.837E+02	-2.548E+02			
Haskell	p	7.039E-46	3.276E-36	5.572E-30	0.000E+00		
	N	5.380E+02	4.200E+02	3.440E+02	5.380E+02		
	W	0.000E+00	0.000E+00	0.000E+00	3.054E+04		
	Z	-1.422E+01	-1.257E+01	-1.137E+01	9.697E+00		
	d	2.511E+00	2.256E+00	2.549E+00	1.041E+00		
	Δ	-1.110E+02	-2.699E+02	-1.751E+02	4.204E-01		
	R	-1.110E+02	-2.671E+02	-1.724E+02	1.366E+00		
Java	p	0.000E+00	7.406E-04	0.000E+00	0.000E+00	0.000E+00	
	N	4.920E+02	3.900E+02	3.140E+02	4.920E+02	4.280E+02	
	W	2.727E+04	2.972E+03	8.158E+03	3.038E+04	2.300E+04	
	Z	1.336E+01	3.374E+00	9.341E+00	1.360E+01	1.268E+01	
	d	2.863E+00	2.489E-01	1.478E+00	3.141E+00	1.617E+00	
	Δ	1.726E+00	1.324E-01	8.141E-01	4.162E+02	3.170E+02	
	R	2.593E+00	1.111E+00	1.745E+00	3.975E+02	3.084E+02	
Python	p	0.000E+00	1.554E-15	0.000E+00	0.000E+00	0.000E+00	7.684E-07
	N	6.860E+02	5.100E+02	4.060E+02	6.960E+02	5.880E+02	5.460E+02
	W	5.897E+04	6.516E+03	1.353E+04	6.073E+04	4.336E+04	2.998E+03
	Z	1.699E+01	7.981E+00	1.165E+01	1.617E+01	1.486E+01	4.943E+00
	d	5.140E+00	8.327E-01	1.571E+00	3.453E+00	1.701E+00	4.218E-01
	Δ	2.037E+00	3.686E-01	1.093E+00	4.770E+02	3.568E+02	3.471E-01
	R	3.007E+00	1.347E+00	2.093E+00	4.726E+02	3.517E+02	1.319E+00

Table 40: Comparison of binary size (by mean) for tasks compiling successfully

Figure 41: Size of binaries (mean) of tasks compiling successfully

Figure 42: Size of binaries (mean) of tasks compiling successfully (normalized horizontal distances)

E. Performance

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	9.150E-04						
	N	2.800E+01						
	W	0.000E+00						
	Z	-3.315E+00						
	d	2.009E+00						
	Δ	-3.714E-02						
	R	-Inf						
F#	p	1.449E-03	1.507E-02					
	N	3.000E+01	2.000E+01					
	W	0.000E+00	2.000E+00					
	Z	-3.185E+00	-2.431E+00					
	d	1.997E+00	1.679E+00					
	Δ	-1.000E-01	-2.515E+00					
	R	-Inf	-3.286E+00					
Go	p	3.173E-01	2.450E-02	6.451E-04				
	N	6.600E+01	3.600E+01	3.400E+01				
	W	1.000E+00	1.370E+02	1.200E+02				
	Z	1.000E+00	2.249E+00	3.412E+00				
	d	2.462E-01	3.290E-01	2.127E+00				
	Δ	7.273E-03	-1.041E+01	1.024E-01				
	R	Inf	-3.672E+01	Inf				
Haskell	p	4.652E-01	4.037E-03	8.791E-04	6.789E-02			
	N	5.400E+01	3.400E+01	3.200E+01	5.600E+01			
	W	7.000E+00	1.370E+02	1.185E+02	0.000E+00			
	Z	7.303E-01	2.875E+00	3.327E+00	-1.826E+00			
	d	3.284E-01	9.493E-02	2.019E+00	1.124E-01			
	Δ	4.037E-02	4.731E-02	9.875E-02	-3.952E-02			
	R	3.733E+01	1.291E+00	1.590E+02	-1.553E+00			
Java	p	4.089E-04	3.113E-03	4.602E-01	1.330E-03	5.756E-04		
	N	4.600E+01	3.000E+01	3.000E+01	5.200E+01	4.400E+01		
	W	2.300E+01	8.000E+00	7.300E+01	5.000E+01	2.100E+01		
	Z	-3.534E+00	-2.956E+00	7.385E-01	-3.210E+00	-3.443E+00		
	d	6.839E-02	1.320E+00	1.570E-01	2.504E-01	2.652E-01		
	Δ	-2.841E-02	-2.255E+00	-2.693E-01	2.211E+00	-4.376E-01		
	R	-1.245E+00	-3.012E+00	-1.189E+00	6.021E+00	-2.384E+00		
Python	p	9.551E-06	1.086E-03	1.708E-03	3.620E-04	1.184E-06	2.832E-05	
	N	6.600E+01	3.400E+01	3.600E+01	7.200E+01	6.200E+01	5.200E+01	
	W	3.200E+01	1.170E+02	1.575E+02	1.010E+02	0.000E+00	3.400E+02	
	Z	-4.427E+00	3.267E+00	3.137E+00	-3.566E+00	-4.858E+00	4.187E+00	
	d	1.421E-01	1.708E+00	1.348E+00	2.357E-01	4.048E-01	2.210E+00	
	Δ	2.225E-01	1.545E+00	6.647E+00	1.583E+01	-3.094E-01	6.384E+00	
	R	2.298E+00	2.568E+00	8.246E+00	7.657E+01	-3.397E+00	6.304E+00	
Ruby	p	1.499E-05	8.060E-01	3.766E-03	2.879E-04	3.509E-05	2.538E-04	9.408E-07
	N	6.200E+01	3.600E+01	3.400E+01	7.200E+01	5.800E+01	5.400E+01	7.800E+01
	W	3.100E+01	4.200E+01	1.240E+02	1.050E+02	2.900E+01	3.410E+02	4.000E+01
	Z	-4.329E+00	-2.456E-01	2.897E+00	-3.626E+00	-4.138E+00	3.658E+00	-4.904E+00
	d	3.103E-02	3.044E-01	1.438E+00	3.408E-01	2.520E-01	7.152E-01	1.454E+00
	Δ	1.258E-02	3.792E+00	2.041E+00	6.671E+00	4.426E+00	2.487E+00	-1.558E+00
	R	1.149E+00	3.482E+00	2.958E+00	2.535E+01	1.832E+01	2.921E+00	-2.499E+00

Table 43: Comparison of performance (by min) for tasks running successfully

Figure 44: Performance (min) of tasks running successfully

Figure 45: Performance (min) of tasks running successfully (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	9.259E-04						
	N	2.800E+01						
	W	0.000E+00						
	Z	-3.312E+00						
	d	2.104E+00						
	Δ	-3.807E-02						
	R	-Inf						
F#	p	1.449E-03	2.174E-02					
	N	3.000E+01	2.000E+01					
	W	0.000E+00	5.000E+00					
	Z	-3.185E+00	-2.295E+00					
	d	1.997E+00	1.597E+00					
	Δ	-1.000E-01	-2.425E+00					
	R	-Inf	-3.038E+00					
Go	p	6.547E-01	8.487E-02	6.451E-04				
	N	6.600E+01	3.600E+01	3.400E+01				
	W	2.000E+00	1.250E+02	1.200E+02				
	Z	4.472E-01	1.723E+00	3.412E+00				
	d	1.441E-01	3.291E-01	2.127E+00				
	Δ	4.545E-03	-1.042E+01	1.024E-01				
	R	2.667E+00	-3.664E+01	Inf				
Haskell	p	9.163E-01	3.996E-03	8.791E-04	1.730E-01			
	N	5.400E+01	3.400E+01	3.200E+01	5.600E+01			
	W	1.100E+01	1.370E+02	1.185E+02	4.000E+00			
	Z	1.051E-01	2.879E+00	3.327E+00	-1.363E+00			
	d	2.371E-01	8.247E-02	2.019E+00	1.156E-01			
	Δ	1.546E+00	4.102E-02	9.875E-02	-4.059E-02			
	R	7.882E+00	1.242E+00	1.590E+02	-1.544E+00			
Java	p	4.060E-04	3.131E-03	4.955E-01	1.337E-03	5.673E-04		
	N	4.600E+01	3.000E+01	3.000E+01	5.200E+01	4.400E+01		
	W	2.300E+01	8.000E+00	7.200E+01	5.000E+01	2.100E+01		
	Z	-3.536E+00	-2.955E+00	6.816E-01	-3.208E+00	-3.447E+00		
	d	7.155E-02	1.323E+00	1.940E-01	2.507E-01	2.627E-01		
	Δ	-2.971E-02	-2.289E+00	-3.434E-01	2.213E+00	-4.332E-01		
	R	-1.256E+00	-2.967E+00	-1.241E+00	6.013E+00	-2.347E+00		
Python	p	9.904E-05	1.260E-03	2.138E-03	2.330E-04	1.006E-06	2.851E-05	
	N	6.600E+01	3.400E+01	3.600E+01	7.200E+01	6.200E+01	5.200E+01	
	W	6.500E+01	1.445E+02	1.560E+02	1.010E+02	0.000E+00	3.400E+02	
	Z	-3.893E+00	3.225E+00	3.070E+00	-3.680E+00	-4.891E+00	4.185E+00	
	d	2.293E-01	1.583E+00	1.323E+00	2.335E-01	2.945E-01	2.193E+00	
	Δ	1.589E+00	1.434E+00	6.529E+00	1.568E+01	-5.703E-01	6.400E+00	
	R	4.851E+00	2.238E+00	7.310E+00	4.455E+01	-2.781E+00	5.909E+00	
Ruby	p	1.665E-05	7.291E-01	5.605E-03	1.372E-03	3.551E-05	2.818E-04	4.448E-06
	N	6.200E+01	3.600E+01	3.400E+01	7.200E+01	5.800E+01	5.400E+01	7.800E+01
	W	3.100E+01	4.700E+01	1.350E+02	1.310E+02	2.900E+01	3.400E+02	6.150E+01
	Z	-4.306E+00	-3.464E-01	2.770E+00	-3.201E+00	-4.135E+00	3.631E+00	-4.589E+00
	d	2.825E-02	3.041E-01	1.362E+00	3.409E-01	2.522E-01	7.172E-01	7.369E-01
	Δ	1.145E-02	3.786E+00	1.968E+00	6.672E+00	4.429E+00	2.494E+00	-1.248E+00
	R	1.134E+00	3.412E+00	2.763E+00	2.527E+01	1.828E+01	2.869E+00	-1.886E+00

Table 46: Comparison of performance (by mean) for tasks running successfully

Figure 47: Performance (mean) of tasks running successfully

Figure 48: Performance (mean) of tasks running successfully (normalized horizontal distances)

F. Scalability

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	1.020E-03						
	N	4.400E+01						
	W	2.100E+01						
	Z	-3.285E+00						
	d	3.279E-01						
	Δ	-1.025E+02						
	R	-6.322E+01						
F#	p	1.206E-02	7.537E-02					
	N	2.400E+01	2.200E+01					
	W	7.000E+00	1.300E+01					
	Z	-2.510E+00	-1.778E+00					
	d	4.530E-01	6.500E-01					
	Δ	-1.903E+02	-4.073E+00					
	R	-9.450E+01	-4.007E+00					
Go	p	3.783E-05	2.027E-02	1.565E-02				
	N	7.400E+01	4.600E+01	2.800E+01				
	W	5.650E+01	1.980E+02	9.100E+01				
	Z	-4.120E+00	2.321E+00	2.417E+00				
	d	4.532E-01	3.380E-01	5.775E-01				
	Δ	-1.490E+01	1.472E+01	3.317E+01				
	R	-1.870E+01	6.555E+00	1.366E+01				
Haskell	p	1.605E-05	8.356E-02	9.292E-01	6.156E-04			
	N	6.400E+01	3.800E+01	2.200E+01	6.400E+01			
	W	1.800E+01	5.200E+01	3.400E+01	6.600E+01			
	Z	-4.314E+00	-1.730E+00	8.891E-02	-3.425E+00			
	d	8.950E-01	2.084E-01	4.241E-01	7.052E-01			
	Δ	-4.728E+01	3.594E+01	9.364E+01	-1.738E+01			
	R	-6.437E+01	2.774E+00	2.904E+01	-1.355E+01			
Java	p	2.506E-05	6.612E-01	1.578E-01	1.349E-02	9.809E-02		
	N	6.800E+01	4.400E+01	2.800E+01	6.800E+01	5.800E+01		
	W	5.100E+01	1.400E+02	7.500E+01	1.530E+02	2.940E+02		
	Z	-4.214E+00	4.383E-01	1.412E+00	-2.470E+00	1.654E+00		
	d	3.740E-01	3.643E-01	4.690E-01	5.626E-01	4.237E-01		
	Δ	-3.879E+01	-4.894E+01	1.909E+01	-7.535E+00	2.039E+01		
	R	-3.369E+01	-1.045E+01	1.400E+01	-4.640E+00	8.727E+00		
Python	p	4.799E-06	2.688E-02	9.375E-01	5.165E-04	8.774E-01	7.880E-02	
	N	7.000E+01	4.000E+01	2.400E+01	7.200E+01	6.000E+01	6.400E+01	
	W	3.000E+01	4.000E+01	3.800E+01	1.120E+02	2.400E+02	1.700E+02	
	Z	-4.573E+00	-2.213E+00	-7.845E-02	-3.472E+00	1.543E-01	-1.758E+00	
	d	7.105E-01	3.360E-01	3.180E-01	7.093E-01	4.084E-01	1.159E-01	
	Δ	-5.481E+01	-9.554E+01	-1.089E+01	-2.348E+01	1.757E+01	1.261E+01	
	R	-4.234E+01	-2.776E+01	-2.174E+00	-9.799E+00	5.667E+00	1.696E+00	
Ruby	p	1.984E-04	4.274E-03	7.537E-01	2.473E-04	3.600E-01	1.251E-02	7.141E-02
	N	6.600E+01	3.800E+01	2.400E+01	6.400E+01	6.000E+01	5.800E+01	6.200E+01
	W	6.500E+01	2.400E+01	3.500E+01	6.800E+01	1.880E+02	1.020E+02	1.560E+02
	Z	-3.721E+00	-2.857E+00	-3.138E-01	-3.665E+00	-9.153E-01	-2.497E+00	-1.803E+00
	d	9.986E-01	3.582E-01	1.129E-01	9.836E-01	2.495E-01	2.039E-01	1.896E-02
	Δ	-4.875E+01	-7.520E+01	2.932E+00	-2.920E+01	1.703E+01	4.224E+01	-1.265E+00
	R	-8.562E+00	-1.156E+01	1.357E+00	-9.696E+00	3.955E+00	2.612E+00	-1.093E+00

Table 49: Comparison of scalability (by min) for tasks running successfully

Figure 50: Scalability (min) of tasks running successfully

Figure 51: Scalability (min) of tasks running successfully (normalized horizontal distances)

Figure 52: Scalability (min) of tasks running successfully (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	4.061E-03						
	N	4.400E+01						
	W	3.800E+01						
	Z	-2.873E+00						
	d	3.274E-01						
	Δ	-1.023E+02						
	R	-3.510E+01						
F#	p	2.218E-03	7.537E-02					
	N	2.400E+01	2.200E+01					
	W	0.000E+00	1.300E+01					
	Z	-3.059E+00	-1.778E+00					
	d	4.573E-01	6.528E-01					
	Δ	-1.920E+02	-4.088E+00					
	R	-9.302E+01	-4.018E+00					
Go	p	5.218E-05	1.779E-01	1.315E-02				
	N	7.400E+01	4.600E+01	2.800E+01				
	W	6.800E+01	1.680E+02	9.200E+01				
	Z	-4.046E+00	1.347E+00	2.480E+00				
	d	4.805E-01	2.749E-01	5.920E-01				
	Δ	-1.577E+01	1.229E+01	3.380E+01				
	R	-1.439E+01	3.051E+00	1.084E+01				
Haskell	p	3.902E-06	1.959E-02	3.739E-01	1.150E-04			
	N	6.400E+01	3.800E+01	2.200E+01	6.400E+01			
	W	4.000E+00	3.700E+01	2.300E+01	4.500E+01			
	Z	-4.617E+00	-2.334E+00	-8.891E-01	-3.857E+00			
	d	8.685E-01	1.171E-02	1.991E-01	5.156E-01			
	Δ	-5.401E+01	2.265E+00	4.857E+01	-3.286E+01			
	R	-2.920E+01	1.042E+00	2.003E+00	-1.619E+01			
Java	p	2.702E-05	8.329E-01	6.404E-02	3.939E-02	3.692E-02		
	N	6.800E+01	4.400E+01	2.800E+01	6.800E+01	5.800E+01		
	W	5.200E+01	1.330E+02	8.200E+01	1.770E+02	3.140E+02		
	Z	-4.197E+00	2.110E-01	1.852E+00	-2.060E+00	2.087E+00		
	d	4.561E-01	3.680E-01	5.048E-01	5.498E-01	4.642E-01		
	Δ	-5.066E+01	-4.941E+01	2.045E+01	-9.282E+00	3.623E+01		
	R	-2.373E+01	-1.021E+01	1.422E+01	-3.835E+00	8.638E+00		
Python	p	1.246E-06	1.713E-03	4.802E-01	1.672E-05	6.583E-01	2.484E-02	
	N	7.000E+01	4.000E+01	2.400E+01	7.200E+01	6.000E+01	6.400E+01	
	W	1.900E+01	2.100E+01	3.000E+01	5.900E+01	2.110E+02	1.440E+02	
	Z	-4.848E+00	-3.136E+00	-7.060E-01	-4.305E+00	-4.422E-01	-2.244E+00	
	d	6.054E-01	5.089E-01	3.430E-01	8.628E-01	5.824E-02	1.762E-01	
	Δ	-1.013E+02	-1.729E+02	-1.163E+01	-3.462E+01	1.261E+01	-3.434E+01	
	R	-4.223E+01	-4.683E+01	-2.062E+00	-1.077E+01	1.256E+00	-2.082E+00	
Ruby	p	5.597E-05	2.902E-03	1.823E-01	1.211E-05	8.130E-01	1.591E-02	1.264E-01
	N	6.600E+01	3.800E+01	2.400E+01	6.400E+01	6.000E+01	5.800E+01	6.200E+01
	W	5.500E+01	2.100E+01	2.200E+01	3.000E+01	2.210E+02	1.060E+02	1.700E+02
	Z	-4.029E+00	-2.978E+00	-1.334E+00	-4.376E+00	-2.365E-01	-2.411E+00	-1.529E+00
	d	1.035E+00	4.273E-01	3.170E-01	8.534E-01	3.905E-01	1.097E-01	7.670E-02
	Δ	-6.826E+01	-9.025E+01	-2.685E+01	-5.062E+01	3.608E+01	2.359E+01	1.514E+01
	R	-1.008E+01	-1.331E+01	-2.786E+00	-1.268E+01	6.832E+00	1.513E+00	1.314E+00

Table 53: Comparison of scalability (by mean) for tasks running successfully

Figure 54: Scalability (mean) of tasks running successfully

Figure 55: Scalability (mean) of tasks running successfully (normalized horizontal distances)

Figure 56: Scalability (mean) of tasks running successfully (normalized horizontal distances)

G. Maximum RAM

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	7.994E-05						
	N	4.400E+01						
	W	5.000E+00						
	Z	-3.945E+00						
	d	2.022E+00						
	Δ	-8.144E+00						
	R	-8.400E+00						
F#	p	6.040E-03	9.925E-03					
	N	2.400E+01	2.200E+01					
	W	4.000E+00	4.000E+00					
	Z	-2.746E+00	-2.578E+00					
	d	7.613E-01	1.045E+00					
	Δ	-3.585E+01	-3.258E+00					
	R	-2.241E+01	-4.102E+00					
Go	p	1.573E-04	7.687E-05	6.319E-03				
	N	7.400E+01	4.600E+01	2.800E+01				
	W	1.010E+02	2.680E+02	9.600E+01				
	Z	-3.779E+00	3.954E+00	2.731E+00				
	d	6.437E-02	3.914E-01	7.883E-01				
	Δ	8.335E-01	2.005E+01	1.780E+01				
	R	1.217E+00	2.039E+01	1.051E+01				
Haskell	p	1.172E-04	8.405E-01	6.188E-02	3.812E-04			
	N	6.400E+01	3.800E+01	2.200E+01	6.400E+01			
	W	5.800E+01	1.000E+02	5.400E+01	7.400E+01			
	Z	-3.852E+00	2.012E-01	1.867E+00	-3.553E+00			
	d	2.874E-01	1.234E-01	6.139E-01	3.144E-01			
	Δ	-7.141E+01	-6.758E-01	9.199E+00	-3.401E+01			
	R	-1.810E+01	-1.225E+00	7.405E+00	-1.861E+01			
Java	p	1.151E-06	7.994E-05	3.305E-01	3.181E-06	7.101E-03		
	N	6.800E+01	4.400E+01	2.800E+01	6.800E+01	5.800E+01		
	W	1.300E+01	5.000E+00	3.700E+01	2.500E+01	9.300E+01		
	Z	-4.864E+00	-3.945E+00	-9.730E-01	-4.659E+00	-2.692E+00		
	d	8.903E-01	1.427E+00	2.782E-01	5.273E-01	6.167E-01		
	Δ	-5.109E+01	-2.484E+00	-1.926E-01	-4.241E+01	-5.429E+00		
	R	-4.137E+01	-3.388E+00	-1.143E+00	-3.504E+01	-3.236E+00		
Python	p	1.056E-06	3.507E-01	3.417E-02	1.257E-05	9.918E-01	5.982E-03	
	N	7.000E+01	4.000E+01	2.400E+01	7.200E+01	6.000E+01	6.400E+01	
	W	1.700E+01	8.000E+01	6.600E+01	5.500E+01	2.330E+02	4.110E+02	
	Z	-4.881E+00	-9.333E-01	2.118E+00	-4.368E+00	1.028E-02	2.749E+00	
	d	3.297E-01	4.450E-01	1.040E-01	4.170E-01	8.660E-03	2.018E-01	
	Δ	-3.075E+01	-3.171E+00	-2.621E+00	-2.265E+01	1.959E-01	1.767E+00	
	R	-2.557E+01	-3.210E+00	-1.352E+00	-1.787E+01	1.029E+00	1.523E+00	
Ruby	p	3.245E-06	2.225E-03	5.303E-01	1.437E-05	4.950E-02	2.218E-01	3.601E-02
	N	6.600E+01	3.800E+01	2.400E+01	6.400E+01	6.000E+01	5.800E+01	6.200E+01
	W	2.000E+01	1.900E+01	4.700E+01	3.200E+01	1.370E+02	2.740E+02	1.410E+02
	Z	-4.655E+00	-3.058E+00	6.276E-01	-4.338E+00	-1.964E+00	1.222E+00	-2.097E+00
	d	4.027E-01	5.246E-01	2.424E-01	5.307E-01	3.014E-01	3.009E-01	6.439E-02
	Δ	-6.022E+01	-5.864E+00	3.374E-01	-4.107E+01	-1.119E+01	1.485E+00	1.342E+00
	R	-4.482E+01	-6.066E+00	1.194E+00	-2.609E+01	-2.461E+00	1.726E+00	1.264E+00

Table 57: Comparison of RAM used (by min) for tasks running successfully

Figure 58: Maximum RAM usage (min) of scalability tasks

Figure 59: Maximum RAM usage (min) of scalability tasks (normalized horizontal distances)

Figure 60: Maximum RAM usage (min) of scalability tasks (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	4.010E-05						
	N	4.400E+01						
	W	0.000E+00						
	Z	-4.107E+00						
	d	2.239E+00						
	Δ	-8.700E+00						
	R	-8.651E+00						
F#	p	4.742E-03	9.925E-03					
	N	2.400E+01	2.200E+01					
	W	3.000E+00	4.000E+00					
	Z	-2.824E+00	-2.578E+00					
	d	7.702E-01	1.046E+00					
	Δ	-3.617E+01	-3.259E+00					
	R	-2.260E+01	-4.103E+00					
Go	p	2.833E-05	1.623E-04	4.286E-03				
	N	7.400E+01	4.600E+01	2.800E+01				
	W	7.400E+01	2.620E+02	9.800E+01				
	Z	-4.187E+00	3.771E+00	2.856E+00				
	d	3.563E-02	3.879E-01	8.040E-01				
	Δ	4.633E-01	1.987E+01	1.807E+01				
	R	1.109E+00	1.215E+01	1.063E+01				
Haskell	p	6.292E-05	4.688E-01	7.537E-02	3.304E-04			
	N	6.400E+01	3.800E+01	2.200E+01	6.400E+01			
	W	5.000E+01	7.700E+01	5.300E+01	7.200E+01			
	Z	-4.002E+00	-7.244E-01	1.778E+00	-3.590E+00			
	d	4.214E-01	1.444E-02	6.087E-01	3.505E-01			
	Δ	-1.441E+02	-7.664E-01	9.119E+00	-7.938E+01			
	R	-3.168E+01	-1.045E+00	7.001E+00	-7.281E+00			
Java	p	6.816E-07	6.922E-04	7.776E-01	9.141E-06	1.072E-01		
	N	6.800E+01	4.400E+01	2.800E+01	6.800E+01	5.800E+01		
	W	7.000E+00	2.200E+01	4.800E+01	3.800E+01	1.430E+02		
	Z	-4.967E+00	-3.393E+00	-2.825E-01	-4.437E+00	-1.611E+00		
	d	9.115E-01	4.139E-01	7.520E-02	5.250E-01	2.382E-02		
	Δ	-5.218E+01	-1.737E+00	7.082E-02	-4.223E+01	6.267E-01		
	R	-4.064E+01	-1.813E+00	1.046E+00	-2.037E+01	1.057E+00		
Python	p	3.506E-07	1.913E-01	5.974E-02	6.050E-06	7.655E-01	4.747E-02	
	N	7.000E+01	4.000E+01	2.400E+01	7.200E+01	6.000E+01	6.400E+01	
	W	4.000E+00	7.000E+01	6.300E+01	4.500E+01	2.180E+02	3.700E+02	
	Z	-5.094E+00	-1.307E+00	1.883E+00	-4.525E+00	-2.982E-01	1.982E+00	
	d	3.715E-01	3.831E-01	1.278E-01	4.468E-01	1.044E-01	6.681E-02	
	Δ	-4.085E+01	-3.914E+00	-3.200E+00	-2.745E+01	3.778E+00	6.154E-01	
	R	-3.206E+01	-2.291E+00	-1.413E+00	-1.195E+01	1.302E+00	1.124E+00	
Ruby	p	5.912E-07	1.124E-02	8.753E-01	1.111E-05	1.846E-01	2.386E-01	2.319E-01
	N	6.600E+01	3.800E+01	2.400E+01	6.400E+01	6.000E+01	5.800E+01	6.200E+01
	W	1.000E+00	3.200E+01	3.700E+01	2.900E+01	1.680E+02	2.720E+02	1.870E+02
	Z	-4.994E+00	-2.535E+00	-1.569E-01	-4.394E+00	-1.327E+00	1.178E+00	-1.195E+00
	d	4.898E-01	5.879E-01	3.501E-01	6.147E-01	1.666E-01	3.979E-02	4.912E-02
	Δ	-7.925E+01	-7.941E+00	-4.416E+00	-4.891E+01	-7.725E+00	-3.747E-01	-1.144E+00
	R	-5.805E+01	-5.333E+00	-2.746E+00	-2.460E+01	-1.468E+00	-1.098E+00	-1.143E+00

Table 61: Comparison of RAM used (by mean) for tasks running successfully

Figure 62: Maximum RAM usage (mean) of scalability tasks

Figure 63: Maximum RAM usage (mean) of scalability tasks (normalized horizontal distances)

Figure 64: Maximum RAM usage (mean) of scalability tasks (normalized horizontal distances)

H. Page faults

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	NaN						
	N	4.400E+01						
	W	NaN						
	Z	NaN						
	d	NaN						
	Δ	NaN						
	R	NaN						
F#	p	4.509E-03	4.509E-03					
	N	2.400E+01	2.200E+01					
	W	0.000E+00	0.000E+00					
	Z	-2.840E+00	-2.840E+00					
	d	6.346E-01	6.668E-01					
	Δ	-5.583E+00	-6.091E+00					
	R	-Inf	-Inf					
Go	p	NaN	NaN	1.847E-03				
	N	7.400E+01	4.600E+01	2.800E+01				
	W	NaN	NaN	7.800E+01				
	Z	NaN	NaN	3.114E+00				
	d	NaN	NaN	6.027E-01				
	Δ	NaN	NaN	4.929E+00				
	R	NaN	NaN	Inf				
Haskell	p	6.051E-07	1.121E-04	5.508E-02	6.568E-07			
	N	6.400E+01	3.800E+01	2.200E+01	6.400E+01			
	W	0.000E+00	0.000E+00	1.150E+01	0.000E+00			
	Z	-4.990E+00	-3.863E+00	-1.918E+00	-4.974E+00			
	d	4.959E+00	5.024E+00	8.042E-01	4.309E+00			
	Δ	-5.719E+00	-5.474E+00	-2.386E+00	-5.969E+00			
	R	-Inf	-Inf	-2.207E+00	-Inf			
Java	p	NaN	NaN	1.847E-03	NaN	2.176E-06		
	N	6.800E+01	4.400E+01	2.800E+01	6.800E+01	5.800E+01		
	W	NaN	NaN	7.800E+01	NaN	4.350E+02		
	Z	NaN	NaN	3.114E+00	NaN	4.736E+00		
	d	NaN	NaN	6.027E-01	NaN	4.501E+00		
	Δ	NaN	NaN	4.929E+00	NaN	5.793E+00		
	R	NaN	NaN	Inf	NaN	Inf		
Python	p	NaN	NaN	4.402E-03	NaN	1.459E-06	NaN	
	N	7.000E+01	4.000E+01	2.400E+01	7.200E+01	6.000E+01	6.400E+01	
	W	NaN	NaN	5.500E+01	NaN	4.650E+02	NaN	
	Z	NaN	NaN	2.848E+00	NaN	4.817E+00	NaN	
	d	NaN	NaN	6.237E-01	NaN	4.207E+00	NaN	
	Δ	NaN	NaN	5.500E+00	NaN	6.000E+00	NaN	
	R	NaN	NaN	Inf	NaN	Inf	NaN	
Ruby	p	NaN	NaN	4.402E-03	NaN	1.282E-06	NaN	NaN
	N	6.600E+01	3.800E+01	2.400E+01	6.400E+01	6.000E+01	5.800E+01	6.200E+01
	W	NaN	NaN	5.500E+01	NaN	4.650E+02	NaN	NaN
	Z	NaN	NaN	2.848E+00	NaN	4.843E+00	NaN	NaN
	d	NaN	NaN	6.237E-01	NaN	4.232E+00	NaN	NaN
	Δ	NaN	NaN	5.500E+00	NaN	5.867E+00	NaN	NaN
	R	NaN	NaN	Inf	NaN	Inf	NaN	NaN

Table 65: Comparison of page faults (by min) for tasks running successfully

Figure 66: Page faults (min) of scalability tasks

Figure 67: Page faults (min) of scalability tasks (normalized horizontal distances)

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	3.173E-01						
	N	4.400E+01						
	W	1.000E+00						
	Z	1.000E+00						
	d	2.194E-01						
	Δ	3.409E-02						
	R	4.000E+00						
F#	p	4.509E-03	4.509E-03					
	N	2.400E+01	2.200E+01					
	W	0.000E+00	0.000E+00					
	Z	-2.840E+00	-2.840E+00					
	d	6.346E-01	6.668E-01					
	Δ	-5.583E+00	-6.091E+00					
	R	-Inf	-Inf					
Go	p	3.173E-01	3.173E-01	1.847E-03				
	N	7.400E+01	4.600E+01	2.800E+01				
	W	1.000E+00	1.000E+00	7.800E+01				
	Z	1.000E+00	1.000E+00	3.114E+00				
	d	2.325E-01	2.949E-01	6.027E-01				
	Δ	2.703E-02	1.087E-02	4.929E+00				
	R	Inf	Inf	Inf				
Haskell	p	7.291E-07	1.253E-04	5.557E-02	7.490E-07			
	N	6.400E+01	3.800E+01	2.200E+01	6.400E+01			
	W	0.000E+00	0.000E+00	1.150E+01	0.000E+00			
	Z	-4.953E+00	-3.835E+00	-1.914E+00	-4.948E+00			
	d	2.834E-01	3.494E-01	8.268E-01	2.851E-01			
	Δ	-4.976E+01	-7.961E+01	-2.443E+00	-5.004E+01			
	R	-1.593E+03	-6.051E+03	-2.236E+00	-Inf			
Java	p	3.173E-01	3.173E-01	1.847E-03	NaN	2.408E-06		
	N	6.800E+01	4.400E+01	2.800E+01	6.800E+01	5.800E+01		
	W	1.000E+00	1.000E+00	7.800E+01	NaN	4.350E+02		
	Z	1.000E+00	1.000E+00	3.114E+00	NaN	4.716E+00		
	d	2.425E-01	3.015E-01	6.027E-01	NaN	2.950E-01		
	Δ	2.941E-02	1.136E-02	4.929E+00	NaN	5.441E+01		
	R	Inf	Inf	Inf	NaN	Inf		
Python	p	6.547E-01	3.173E-01	4.402E-03	3.173E-01	1.635E-06	3.173E-01	
	N	7.000E+01	4.000E+01	2.400E+01	7.200E+01	6.000E+01	6.400E+01	
	W	1.000E+00	0.000E+00	5.500E+01	0.000E+00	4.650E+02	0.000E+00	
	Z	-4.472E-01	-1.000E+00	2.848E+00	-1.000E+00	4.794E+00	-1.000E+00	
	d	2.211E-01	1.414E-01	6.237E-01	2.357E-01	2.921E-01	2.500E-01	
	Δ	-2.000E-01	-1.250E-02	5.500E+00	-1.389E-02	5.296E+01	-1.562E-02	
	R	-8.000E+00	-2.000E+00	Inf	-Inf	3.178E+03	-Inf	
Ruby	p	3.173E-01	3.173E-01	4.402E-03	NaN	1.586E-06	NaN	3.173E-01
	N	6.600E+01	3.800E+01	2.400E+01	6.400E+01	6.000E+01	5.800E+01	6.200E+01
	W	1.000E+00	1.000E+00	5.500E+01	NaN	4.650E+02	NaN	1.000E+00
	Z	1.000E+00	1.000E+00	2.848E+00	NaN	4.800E+00	NaN	1.000E+00
	d	2.462E-01	3.244E-01	6.237E-01	NaN	2.914E-01	NaN	2.540E-01
	Δ	3.030E-02	1.316E-02	5.500E+00	NaN	5.284E+01	NaN	1.613E-02
	R	Inf	Inf	Inf	NaN	Inf	NaN	Inf

Table 68: Comparison of page faults (by mean) for tasks running successfully

Figure 69: Page faults (mean) of scalability tasks

Figure 70: Page faults (mean) of scalability tasks (normalized horizontal distances)

I. Timeout analysis

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	4.550E-02						
	N	5.400E+01						
	W	0.000E+00						
	Z	-2.000E+00						
	d	4.760E-01						
	Δ	-1.481E-01						
	R	-5.000E+00						
F#	p	4.550E-02	3.173E-01					
	N	3.400E+01	3.400E+01					
	W	0.000E+00	2.500E+00					
	Z	-2.000E+00	-1.000E+00					
	d	6.295E-01	2.717E-01					
	Δ	-2.353E-01	-1.176E-01					
	R	-5.000E+00	-1.667E+00					
Go	p	1.573E-01	8.326E-02	4.550E-02				
	N	8.000E+01	5.600E+01	3.800E+01				
	W	0.000E+00	6.000E+00	1.000E+01				
	Z	-1.414E+00	1.732E+00	2.000E+00				
	d	2.280E-01	3.224E-01	5.869E-01				
	Δ	-5.000E-02	1.071E-01	2.105E-01				
	R	-3.000E+00	2.500E+00	5.000E+00				
Haskell	p	5.878E-02	1.000E+00	2.568E-01	5.878E-02			
	N	7.800E+01	5.000E+01	3.600E+01	7.800E+01			
	W	4.000E+00	1.050E+01	2.000E+01	4.000E+00			
	Z	-1.890E+00	0.000E+00	1.134E+00	-1.890E+00			
	d	4.543E-01	0.000E+00	4.186E-01	4.543E-01			
	Δ	-1.282E-01	0.000E+00	1.667E-01	-1.282E-01			
	R	-6.000E+00	0.000E+00	2.500E+00	-6.000E+00			
Java	p	3.173E-01	1.573E-01	8.326E-02	3.173E-01	2.568E-01		
	N	6.800E+01	5.200E+01	3.800E+01	6.800E+01	6.800E+01		
	W	0.000E+00	3.000E+00	6.000E+00	0.000E+00	2.000E+01		
	Z	-1.000E+00	1.414E+00	1.732E+00	-1.000E+00	1.134E+00		
	d	1.415E-01	2.378E-01	4.049E-01	1.415E-01	2.891E-01		
	Δ	-2.941E-02	7.692E-02	1.579E-01	-2.941E-02	8.824E-02		
	R	-2.000E+00	2.000E+00	2.500E+00	-2.000E+00	2.500E+00		
Python	p	8.326E-02	6.547E-01	1.797E-01	1.573E-01	3.173E-01	3.173E-01	
	N	7.800E+01	5.200E+01	3.600E+01	7.800E+01	8.000E+01	7.000E+01	
	W	0.000E+00	9.000E+00	1.200E+01	0.000E+00	3.000E+01	2.500E+00	
	Z	-1.732E+00	4.472E-01	1.342E+00	-1.414E+00	1.000E+00	-1.000E+00	
	d	3.139E-01	1.107E-01	4.186E-01	2.311E-01	2.163E-01	2.022E-01	
	Δ	-7.692E-02	3.846E-02	1.667E-01	-5.128E-02	7.500E-02	-5.714E-02	
	R	-4.000E+00	1.333E+00	2.500E+00	-3.000E+00	1.750E+00	-2.000E+00	
Ruby	p	1.025E-01	4.142E-01	4.550E-02	7.055E-01	1.797E-01	3.173E-01	7.055E-01
	N	7.800E+01	5.200E+01	3.400E+01	7.800E+01	7.600E+01	6.800E+01	7.800E+01
	W	3.500E+00	1.400E+01	1.000E+01	1.200E+01	1.200E+01	2.500E+00	1.200E+01
	Z	-1.633E+00	8.165E-01	2.000E+00	-3.780E-01	1.342E+00	-1.000E+00	-3.780E-01
	d	3.872E-01	2.103E-01	6.295E-01	8.864E-02	2.228E-01	2.054E-01	7.929E-02
	Δ	-1.026E-01	7.692E-02	2.353E-01	-2.564E-02	7.895E-02	-5.882E-02	-2.564E-02
	R	-5.000E+00	1.667E+00	5.000E+00	-1.333E+00	1.750E+00	-2.000E+00	-1.250E+00

Table 71: Comparison of time-outs (3 minutes) on scalability tasks

Figure 72: Timeout analysis of scalability tasks

Figure 73: Timeout analysis of scalability tasks (normalized horizontal distances)

J. Number of solutions

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	0.000E+00						
	N	1.488E+03						
	W	2.168E+04						
	Z	9.277E+00						
	d	2.975E-01						
	Δ	2.285E-01						
	R	1.480E+00						
F#	p	0.000E+00	1.791E-09					
	N	1.488E+03	1.488E+03					
	W	3.365E+04	1.177E+04					
	Z	1.245E+01	6.016E+00					
	d	5.015E-01	1.992E-01					
	Δ	3.629E-01	1.344E-01					
	R	2.063E+00	1.394E+00					
Go	p	1.348E-01	4.389E-13	1.069E-30				
	N	1.488E+03	1.488E+03	1.488E+03				
	W	9.416E+03	6.702E+03	2.973E+03				
	Z	1.495E+00	-7.243E+00	-1.152E+01				
	d	4.663E-02	2.622E-01	4.758E-01				
	Δ	3.629E-02	-1.922E-01	-3.266E-01				
	R	1.054E+00	-1.404E+00	-1.957E+00				
Haskell	p	4.101E-01	5.660E-13	1.560E-29	6.990E-01			
	N	1.488E+03	1.488E+03	1.488E+03	1.488E+03			
	W	1.111E+04	4.853E+03	2.518E+03	1.086E+04			
	Z	8.236E-01	-7.208E+00	-1.128E+01	-3.866E-01			
	d	7.454E-03	2.570E-01	4.324E-01	3.391E-02			
	Δ	6.720E-03	-2.218E-01	-3.562E-01	-2.957E-02			
	R	1.010E+00	-1.466E+00	-2.043E+00	-1.044E+00			
Java	p	8.732E-06	1.621E-07	2.044E-23	1.997E-03	8.913E-04		
	N	1.488E+03	1.488E+03	1.488E+03	1.488E+03	1.488E+03		
	W	1.440E+04	6.256E+03	3.726E+03	1.349E+04	1.215E+04		
	Z	4.446E+00	-5.238E+00	-9.971E+00	3.091E+00	3.323E+00		
	d	1.294E-01	1.612E-01	3.571E-01	8.820E-02	1.089E-01		
	Δ	1.048E-01	-1.237E-01	-2.581E-01	6.855E-02	9.812E-02		
	R	1.175E+00	-1.260E+00	-1.756E+00	1.114E+00	1.164E+00		
Python	p	3.717E-18	4.154E-41	8.135E-53	3.619E-20	1.041E-17	4.076E-29	
	N	1.488E+03	1.488E+03	1.488E+03	1.488E+03	1.488E+03	1.488E+03	
	W	6.380E+03	3.192E+03	1.192E+03	5.853E+03	6.483E+03	3.950E+03	
	Z	-8.687E+00	-1.343E+01	-1.530E+01	-9.199E+00	-8.569E+00	-1.120E+01	
	d	2.978E-01	5.134E-01	6.534E-01	3.368E-01	2.870E-01	3.906E-01	
	Δ	-3.374E-01	-5.659E-01	-7.003E-01	-3.737E-01	-3.441E-01	-4.422E-01	
	R	-1.479E+00	-2.189E+00	-3.051E+00	-1.559E+00	-1.493E+00	-1.738E+00	
Ruby	p	2.206E-02	6.116E-26	4.102E-40	8.831E-04	4.289E-03	8.063E-11	3.735E-13
	N	1.488E+03	1.488E+03	1.488E+03	1.488E+03	1.488E+03	1.488E+03	1.488E+03
	W	1.022E+04	3.386E+03	2.742E+03	8.373E+03	9.946E+03	5.436E+03	2.327E+04
	Z	-2.289E+00	-1.053E+01	-1.326E+01	-3.325E+00	-2.856E+00	-6.499E+00	7.265E+00
	d	8.581E-02	3.472E-01	5.213E-01	1.291E-01	8.564E-02	1.994E-01	2.122E-01
	Δ	-7.930E-02	-3.078E-01	-4.422E-01	-1.156E-01	-8.602E-02	-1.841E-01	2.581E-01
	R	-1.113E+00	-1.647E+00	-2.295E+00	-1.173E+00	-1.123E+00	-1.307E+00	1.329E+00

Table 74: Comparison of number of solutions per task

Figure 75: Number of solutions per task

Figure 76: Number of solutions per task (normalized horizontal distances)

K. Compilation and execution statistics

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	2.210E-02						
	N	8.780E+02						
	U	8.511E+04						
	Z	-2.289E+00						
	d	1.714E-01						
	Δ	-3.091E-01						
F#	p	3.302E-06	3.089E-09					
	N	7.780E+02	6.080E+02					
	U	7.822E+04	5.583E+04					
	Z	4.651E+00	5.927E+00					
	d	2.440E-01	4.392E-01					
	Δ	4.210E-01	7.301E-01					
Go	p	3.285E-04	6.224E-07	8.050E-02				
	N	1.021E+03	8.510E+02	7.510E+02				
	U	1.446E+05	1.030E+05	5.936E+04				
	Z	3.592E+00	4.984E+00	-1.748E+00				
	d	8.553E-02	2.607E-01	1.576E-01				
	Δ	1.543E-01	4.634E-01	-2.667E-01				
Haskell	p	5.511E-13	2.488E-05	6.446E-21	1.741E-20			
	N	1.043E+03	8.730E+02	7.730E+02	1.016E+03			
	U	1.033E+05	7.753E+04	4.107E+04	9.008E+04			
	Z	-7.212E+00	-4.216E+00	-9.382E+00	-9.277E+00			
	d	4.832E-01	3.183E-01	7.508E-01	5.734E-01			
	Δ	-8.882E-01	-5.791E-01	-1.309E+00	-1.043E+00			
Java	p	9.520E-01	2.687E-01	6.655E-06	5.570E-04	2.318E-06		
	N	9.700E+02	8.000E+02	7.000E+02	9.430E+02	9.650E+02		
	U	1.166E+05	8.213E+04	4.719E+04	9.902E+04	1.345E+05		
	Z	-6.024E-02	1.106E+00	-4.504E+00	-3.452E+00	4.724E+00		
	d	1.628E-01	6.199E-03	3.848E-01	2.430E-01	2.851E-01		
	Δ	-3.214E-01	-1.229E-02	-7.423E-01	-4.757E-01	5.668E-01		
Python	p	4.132E-03	4.639E-08	1.521E-02	5.835E-01	0.000E+00	1.935E-04	
	N	1.299E+03	1.129E+03	1.029E+03	1.272E+03	1.294E+03	1.221E+03	
	U	2.194E+05	1.610E+05	9.040E+04	1.897E+05	2.675E+05	1.914E+05	
	Z	2.868E+00	5.465E+00	-2.427E+00	-5.482E-01	1.133E+01	3.727E+00	
	d	1.572E-01	3.685E-01	1.239E-01	5.988E-02	7.173E-01	3.359E-01	
	Δ	2.469E-01	5.560E-01	-1.741E-01	9.258E-02	1.135E+00	5.683E-01	
Ruby	p	2.887E-14	0.000E+00	1.889E-01	1.254E-04	0.000E+00	2.491E-13	8.535E-08
	N	1.105E+03	9.350E+02	8.350E+02	1.078E+03	1.100E+03	1.027E+03	1.356E+03
	U	1.850E+05	1.333E+05	7.676E+04	1.588E+05	2.172E+05	1.564E+05	2.549E+05
	Z	7.603E+00	9.398E+00	1.314E+00	3.835E+00	1.456E+01	7.319E+00	5.355E+00
	d	4.150E-01	6.588E-01	1.620E-01	3.196E-01	9.937E-01	5.731E-01	2.957E-01
	Δ	6.269E-01	9.359E-01	2.059E-01	4.725E-01	1.515E+00	9.482E-01	3.800E-01

Table 77: Unpaired comparisons of compilation status

Figure 78: Comparisons of compilation status

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	7.612E-01						
	N	7.490E+02						
	U	6.718E+04						
	Z	-3.040E-01						
	d	4.167E-02						
	Δ	4.716E-02						
F#	p	1.522E-01	1.134E-01					
	N	6.780E+02	5.430E+02					
	U	5.469E+04	3.835E+04					
	Z	1.432E+00	1.583E+00					
	d	1.351E-01	1.067E-01					
	Δ	1.547E-01	1.075E-01					
Go	p	3.141E-07	8.096E-08	2.259E-03				
	N	8.690E+02	7.340E+02	6.630E+02				
	U	1.070E+05	7.566E+04	5.478E+04				
	Z	5.115E+00	5.365E+00	3.054E+00				
	d	3.887E-01	4.017E-01	2.810E-01				
	Δ	3.876E-01	3.404E-01	2.329E-01				
Haskell	p	1.594E-02	9.047E-03	5.384E-01	4.348E-03			
	N	8.680E+02	7.330E+02	6.620E+02	8.530E+02			
	U	1.005E+05	7.070E+04	5.125E+04	8.463E+04			
	Z	2.410E+00	2.610E+00	6.152E-01	-2.852E+00			
	d	2.001E-01	1.760E-01	6.403E-02	2.092E-01			
	Δ	2.156E-01	1.685E-01	6.097E-02	-1.719E-01			
Java	p	7.743E-01	9.169E-01	1.015E-01	1.165E-07	7.911E-03		
	N	7.810E+02	6.460E+02	5.750E+02	7.660E+02	7.650E+02		
	U	7.424E+04	5.185E+04	3.764E+04	6.169E+04	6.644E+04		
	Z	-2.868E-01	-1.044E-01	-1.637E+00	-5.299E+00	-2.656E+00		
	d	9.449E-03	3.386E-02	1.337E-01	4.124E-01	2.022E-01		
	Δ	1.107E-02	-3.610E-02	-1.436E-01	-3.765E-01	-2.046E-01		
Python	p	4.197E-02	6.413E-02	1.566E-03	5.994E-14	1.639E-06	9.879E-02	
	N	1.193E+03	1.058E+03	9.870E+02	1.178E+03	1.177E+03	1.090E+03	
	U	1.568E+05	1.087E+05	7.924E+04	1.295E+05	1.396E+05	1.210E+05	
	Z	-2.034E+00	-1.851E+00	-3.162E+00	-7.508E+00	-4.793E+00	-1.651E+00	
	d	1.120E-01	1.571E-01	2.462E-01	4.876E-01	3.117E-01	1.237E-01	
	Δ	-1.375E-01	-1.847E-01	-2.922E-01	-5.251E-01	-3.531E-01	-1.486E-01	
Ruby	p	3.597E-01	3.048E-01	4.397E-01	5.695E-06	8.662E-02	2.544E-01	1.201E-03
	N	1.005E+03	8.700E+02	7.990E+02	9.900E+02	9.890E+02	9.020E+02	1.314E+03
	U	1.275E+05	8.911E+04	6.480E+04	1.069E+05	1.146E+05	9.860E+04	2.283E+05
	Z	9.159E-01	1.026E+00	-7.727E-01	-4.537E+00	-1.714E+00	1.140E+00	3.239E+00
	d	6.754E-02	2.850E-02	7.204E-02	3.266E-01	1.349E-01	6.021E-02	1.824E-01
	Δ	7.753E-02	3.037E-02	-7.714E-02	-3.100E-01	-1.381E-01	6.647E-02	2.150E-01

Table 79: Unpaired comparisons of running status

Figure 80: Comparisons of running status

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	5.525E-05						
	N	7.490E+02						
	U	5.690E+04						
	Z	-4.032E+00						
	d	3.231E-01						
	Δ	-5.362E-01						
F#	p	2.097E-04	1.676E-11					
	N	6.780E+02	5.430E+02					
	U	6.023E+04	4.738E+04					
	Z	3.707E+00	6.732E+00					
	d	3.497E-01	6.970E-01					
	Δ	5.022E-01	1.038E+00					
Go	p	3.584E-09	0.000E+00	1.479E-01				
	N	8.690E+02	7.340E+02	6.630E+02				
	U	1.132E+05	8.916E+04	5.315E+04				
	Z	5.902E+00	9.507E+00	1.447E+00				
	d	4.174E-01	7.971E-01	6.011E-02				
	Δ	5.655E-01	1.102E+00	6.338E-02				
Haskell	p	1.604E-11	5.731E-02	3.464E-20	7.569E-38			
	N	8.680E+02	7.330E+02	6.620E+02	8.530E+02			
	U	7.077E+04	6.034E+04	3.018E+04	4.915E+04			
	Z	-6.738E+00	-1.901E+00	-9.204E+00	-1.286E+01			
	d	4.712E-01	1.379E-01	8.549E-01	9.566E-01			
	Δ	-7.715E-01	-2.354E-01	-1.274E+00	-1.337E+00			
Java	p	8.076E-01	2.447E-04	2.032E-03	8.684E-08	5.558E-09		
	N	7.810E+02	6.460E+02	5.750E+02	7.660E+02	7.650E+02		
	U	7.561E+04	6.001E+04	3.474E+04	5.878E+04	8.859E+04		
	Z	2.435E-01	3.668E+00	-3.086E+00	-5.352E+00	5.830E+00		
	d	3.523E-02	2.845E-01	3.919E-01	4.659E-01	4.315E-01		
	Δ	-5.677E-02	4.794E-01	-5.589E-01	-6.223E-01	7.148E-01		
Python	p	8.450E-04	9.472E-03	1.306E-11	1.560E-23	5.476E-10	1.853E-02	
	N	1.193E+03	1.058E+03	9.870E+02	1.178E+03	1.177E+03	1.090E+03	
	U	1.480E+05	1.263E+05	6.459E+04	1.096E+05	1.929E+05	1.168E+05	
	Z	-3.338E+00	2.595E+00	-6.768E+00	-9.998E+00	6.205E+00	-2.355E+00	
	d	1.590E-01	1.916E-01	5.404E-01	6.093E-01	3.452E-01	1.212E-01	
	Δ	-2.400E-01	2.961E-01	-7.422E-01	-8.056E-01	5.315E-01	-1.833E-01	
Ruby	p	2.424E-02	4.044E-11	4.377E-02	2.010E-05	0.000E+00	3.572E-02	1.594E-09
	N	1.005E+03	8.700E+02	7.990E+02	9.900E+02	9.890E+02	9.020E+02	1.314E+03
	U	1.336E+05	1.076E+05	6.123E+04	1.043E+05	1.611E+05	1.024E+05	2.492E+05
	Z	2.253E+00	6.602E+00	-2.016E+00	-4.264E+00	1.006E+01	2.100E+00	6.035E+00
	d	1.640E-01	5.263E-01	2.178E-01	2.780E-01	6.855E-01	2.046E-01	3.427E-01
	Δ	2.349E-01	7.711E-01	-2.673E-01	-3.306E-01	1.006E+00	2.917E-01	4.750E-01

Table 81: Unpaired comparisons of combined compilation and running

Figure 82: Comparisons of combined compilation and running status

LANGUAGE	MEASURE	C	C#	F#	Go	Haskell	Java	Python
C#	p	3.687E-02						
	N	6.370E+02						
	U	5.060E+04						
	Z	2.087E+00						
	d	1.703E-01						
	Δ	5.215E-02						
F#	p	5.047E-01	2.038E-01					
	N	6.060E+02	4.610E+02					
	U	4.280E+04	2.555E+04					
	Z	6.671E-01	-1.271E+00					
	d	5.662E-02	1.187E-01					
	Δ	1.834E-02	-3.381E-02					
Go	p	1.944E-08	1.129E-03	4.138E-06				
	N	7.800E+02	6.350E+02	6.040E+02				
	U	8.402E+04	5.036E+04	4.543E+04				
	Z	5.617E+00	3.256E+00	4.604E+00				
	d	4.104E-01	2.673E-01	3.980E-01				
	Δ	1.048E-01	5.261E-02	8.641E-02				
Haskell	p	5.836E-03	7.483E-01	8.328E-02	1.787E-03			
	N	7.670E+02	6.220E+02	5.910E+02	7.650E+02			
	U	7.783E+04	4.656E+04	4.206E+04	6.977E+04			
	Z	2.757E+00	3.209E-01	1.732E+00	-3.123E+00			
	d	2.000E-01	2.630E-02	1.483E-01	2.272E-01			
	Δ	5.883E-02	6.681E-03	4.049E-02	-4.592E-02			
Java	p	3.860E-01	6.292E-03	1.732E-01	4.081E-10	5.300E-04		
	N	6.880E+02	5.430E+02	5.120E+02	6.860E+02	6.730E+02		
	U	5.674E+04	3.379E+04	3.061E+04	5.040E+04	5.128E+04		
	Z	-8.668E-01	-2.732E+00	-1.362E+00	-6.251E+00	-3.465E+00		
	d	6.671E-02	2.370E-01	1.221E-01	4.957E-01	2.712E-01		
	Δ	-2.283E-02	-7.498E-02	-4.117E-02	-1.276E-01	-8.166E-02		
Python	p	7.520E-04	1.837E-06	9.473E-04	2.462E-17	1.885E-09	3.066E-02	
	N	1.067E+03	9.220E+02	8.910E+02	1.065E+03	1.052E+03	9.730E+02	
	U	1.214E+05	7.201E+04	6.539E+04	1.070E+05	1.092E+05	9.447E+04	
	Z	-3.370E+00	-4.771E+00	-3.306E+00	-8.470E+00	-6.007E+00	-2.161E+00	
	d	2.152E-01	3.595E-01	2.603E-01	5.579E-01	3.931E-01	1.508E-01	
	Δ	-8.178E-02	-1.339E-01	-1.001E-01	-1.865E-01	-1.406E-01	-5.895E-02	
Ruby	p	5.891E-01	9.674E-03	2.601E-01	6.770E-10	7.194E-04	6.778E-01	1.847E-03
	N	9.070E+02	7.620E+02	7.310E+02	9.050E+02	8.920E+02	8.130E+02	1.192E+03
	U	9.964E+04	5.938E+04	5.377E+04	8.862E+04	9.011E+04	7.743E+04	1.865E+05
	Z	-5.401E-01	-2.587E+00	-1.126E+00	-6.171E+00	-3.382E+00	4.154E-01	3.114E+00
	d	3.620E-02	2.012E-01	9.143E-02	4.232E-01	2.307E-01	3.024E-02	1.827E-01
	Δ	-1.228E-02	-6.443E-02	-3.062E-02	-1.170E-01	-7.111E-02	1.055E-02	6.950E-02

Table 83: Unpaired comparisons of runtime fault proneness

Figure 84: Comparisons of fault proneness (based on exit status) of solutions that compile correctly and do not timeout

	tasks	solutions	make ok	none	libs	merge	patch	fix	make ko
C	398	524	0.85	0.68	0.21	0.03	0.06	0.02	0.15
C#	288	354	0.90	0.58	0.08	0.00	0.34	0.00	0.10
F#	216	254	0.95	0.82	0.00	0.00	0.18	0.00	0.05
Go	392	497	0.89	0.82	0.00	0.01	0.17	0.00	0.11
Haskell	358	519	0.84	0.47	0.00	0.06	0.47	0.00	0.16
Java	344	446	0.78	0.82	0.01	0.02	0.10	0.04	0.22
Python	415	775	1.00	0.67	0.00	0.05	0.28	0.00	0.00
Ruby	391	581	1.00	0.80	0.00	0.03	0.17	0.00	0.00
TOTAL	442	3950	0.91	0.70	0.03	0.03	0.22	0.01	0.09

Table 85: Statistics about the compilation process: columns **make ok** and **make ko** report percentages relative to **solutions** for each language; the columns in between report percentages relative to **make ok** for each language

	tasks	solutions	run ok	stderr	timeout	error	crash
C	351	442	0.76	0.01	0.11	0.06	0.05
C#	253	307	0.73	0.01	0.20	0.06	0.00
F#	201	236	0.81	0.01	0.09	0.10	0.00
Go	349	427	0.89	0.01	0.09	0.01	0.00
Haskell	298	426	0.82	0.00	0.12	0.06	0.00
Java	275	339	0.75	0.02	0.11	0.13	0.00
Python	399	751	0.71	0.01	0.10	0.19	0.00
Ruby	378	563	0.79	0.01	0.08	0.12	0.00
TOTAL	422	3491	0.77	0.01	0.11	0.10	0.01

Table 86: Statistics about the running process: all columns *other than* **tasks** and **solutions** report percentages relative to **solutions** for each language

	tasks	solutions	error	run ok
C	313	391	0.13	0.87
C#	203	246	0.07	0.93
F#	182	215	0.11	0.89
Go	317	389	0.02	0.98
Haskell	264	376	0.07	0.93
Java	247	297	0.15	0.85
Python	357	676	0.21	0.79
Ruby	349	516	0.14	0.86
TOTAL	422	3106	0.12	0.88

Table 87: Statistics about fault proneness: columns **error** and **run ok** report percentages relative to **solutions** for each language

C: incomplete (27 solutions), snippet (21 solutions), nomain (21 solutions), nonstandard libs (17 solutions), other (12 solutions), unsupported dialect (2 solutions) C#: nomain (24 solutions), nonstandard libs (17 solutions), snippet (14 solutions), bug (3 solutions), filename (1 solutions), other (1 solutions) F#: nonstandard libs (10 solutions), bug (2 solutions), other (1 solutions) Go: nonstandard libs (41 solutions), snippet (9 solutions), other (5 solutions), bug (2 solutions), nomain (0 solutions) Haskell: nomain (45 solutions), nonstandard libs (30 solutions), snippet (20 solutions), bug (18 solutions), other (16 solutions) Java: nomain (60 solutions), snippet (56 solutions), other (24 solutions), nonstandard libs (13 solutions), incomplete (6 solutions) Python: other (2 solutions) Ruby: other (519 solutions), nonstandard libs (35 solutions), merge (9 solutions), package (7 solutions), bug (6 solutions), snippet (2 solutions), abort (1 solutions), incomplete (1 solutions), require (1 solutions)

Figure 88: Lines of code (min) of tasks compiling successfully (C vs. other languages)

Figure 89: Lines of code (min) of tasks compiling successfully (C vs. other languages)

Figure 90: Lines of code (min) of tasks compiling successfully (C# vs. other languages)

Figure 91: Lines of code (min) of tasks compiling successfully (C# vs. other languages)

Figure 92: Lines of code (min) of tasks compiling successfully (F# vs. other languages)

Figure 93: Lines of code (min) of tasks compiling successfully (F# vs. other languages)

Figure 94: Lines of code (min) of tasks compiling successfully (Go vs. other languages)

Figure 95: Lines of code (min) of tasks compiling successfully (Go vs. other languages)

Figure 96: Lines of code (min) of tasks compiling successfully (Haskell vs. other languages)

Figure 97: Lines of code (min) of tasks compiling successfully (Haskell vs. other languages)

Figure 98: Lines of code (min) of tasks compiling successfully (Java vs. other languages)

Figure 99: Lines of code (min) of tasks compiling successfully (Java vs. other languages)

Figure 100: Lines of code (min) of tasks compiling successfully (Python vs. other languages)

Figure 101: Lines of code (min) of tasks compiling successfully (Python vs. other languages)

Figure 102: Lines of code (min) of tasks compiling successfully (all languages)

Figure 103: Lines of code (mean) of tasks compiling successfully (C vs. other languages)

Figure 104: Lines of code (mean) of tasks compiling successfully (C vs. other languages)

Figure 105: Lines of code (mean) of tasks compiling successfully (C# vs. other languages)

Figure 106: Lines of code (mean) of tasks compiling successfully (C# vs. other languages)

Figure 107: Lines of code (mean) of tasks compiling successfully (F# vs. other languages)

Figure 108: Lines of code (mean) of tasks compiling successfully (F# vs. other languages)

Figure 109: Lines of code (mean) of tasks compiling successfully (Go vs. other languages)

Figure 110: Lines of code (mean) of tasks compiling successfully (Go vs. other languages)

Figure 111: Lines of code (mean) of tasks compiling successfully (Haskell vs. other languages)

Figure 112: Lines of code (mean) of tasks compiling successfully (Haskell vs. other languages)

Figure 113: Lines of code (mean) of tasks compiling successfully (Java vs. other languages)

Figure 114: Lines of code (mean) of tasks compiling successfully (Java vs. other languages)

Figure 115: Lines of code (mean) of tasks compiling successfully (Python vs. other languages)

Figure 116: Lines of code (mean) of tasks compiling successfully (Python vs. other languages)

Figure 117: Lines of code (mean) of tasks compiling successfully (all languages)

Figure 118: Lines of code (min) of all tasks (C vs. other languages)

Figure 119: Lines of code (min) of all tasks (C vs. other languages)

Figure 120: Lines of code (min) of all tasks (C# vs. other languages)

Figure 121: Lines of code (min) of all tasks (C# vs. other languages)

Figure 122: Lines of code (min) of all tasks (F# vs. other languages)

Figure 123: Lines of code (min) of all tasks (F# vs. other languages)

Figure 124: Lines of code (min) of all tasks (Go vs. other languages)

Figure 125: Lines of code (min) of all tasks (Go vs. other languages)

Figure 126: Lines of code (min) of all tasks (Haskell vs. other languages)

Figure 127: Lines of code (min) of all tasks (Haskell vs. other languages)

Figure 128: Lines of code (min) of all tasks (Java vs. other languages)

Figure 129: Lines of code (min) of all tasks (Java vs. other languages)

Figure 130: Lines of code (min) of all tasks (Python vs. other languages)

Figure 131: Lines of code (min) of all tasks (Python vs. other languages)

Figure 132: Lines of code (min) of all tasks (all languages)

Figure 133: Lines of code (mean) of all tasks (C vs. other languages)

Figure 134: Lines of code (mean) of all tasks (C vs. other languages)

Figure 135: Lines of code (mean) of all tasks (C# vs. other languages)

Figure 136: Lines of code (mean) of all tasks (C# vs. other languages)

Figure 137: Lines of code (mean) of all tasks (F# vs. other languages)

Figure 138: Lines of code (mean) of all tasks (F# vs. other languages)

Figure 139: Lines of code (mean) of all tasks (Go vs. other languages)

Figure 140: Lines of code (mean) of all tasks (Go vs. other languages)

Figure 141: Lines of code (mean) of all tasks (Haskell vs. other languages)

Figure 142: Lines of code (mean) of all tasks (Haskell vs. other languages)

Figure 143: Lines of code (mean) of all tasks (Java vs. other languages)

Figure 144: Lines of code (mean) of all tasks (Java vs. other languages)

Figure 145: Lines of code (mean) of all tasks (Python vs. other languages)

Figure 146: Lines of code (mean) of all tasks (Python vs. other languages)

Figure 147: Lines of code (mean) of all tasks (all languages)

Figure 148: Comments per line of code (min) of all tasks (C vs. other languages)

Figure 149: Comments per line of code (min) of all tasks (C vs. other languages)

Figure 150: Comments per line of code (min) of all tasks (C# vs. other languages)

Figure 151: Comments per line of code (min) of all tasks (C# vs. other languages)

Figure 152: Comments per line of code (min) of all tasks (F# vs. other languages)

Figure 153: Comments per line of code (min) of all tasks (F# vs. other languages)

Figure 154: Comments per line of code (min) of all tasks (Go vs. other languages)

Figure 155: Comments per line of code (min) of all tasks (Go vs. other languages)

Figure 156: Comments per line of code (min) of all tasks (Haskell vs. other languages)

Figure 157: Comments per line of code (min) of all tasks (Haskell vs. other languages)

Figure 158: Comments per line of code (min) of all tasks (Java vs. other languages)

Figure 159: Comments per line of code (min) of all tasks (Java vs. other languages)

Figure 160: Comments per line of code (min) of all tasks (Python vs. other languages)

Figure 161: Comments per line of code (min) of all tasks (Python vs. other languages)

Figure 162: Comments per line of code (min) of all tasks (all languages)

Figure 163: Comments per line of code (mean) of all tasks (C vs. other languages)

Figure 164: Comments per line of code (mean) of all tasks (C vs. other languages)

Figure 165: Comments per line of code (mean) of all tasks (C# vs. other languages)

Figure 166: Comments per line of code (mean) of all tasks (C# vs. other languages)

Figure 167: Comments per line of code (mean) of all tasks (F# vs. other languages)

Figure 168: Comments per line of code (mean) of all tasks (F# vs. other languages)

Figure 169: Comments per line of code (mean) of all tasks (Go vs. other languages)

Figure 170: Comments per line of code (mean) of all tasks (Go vs. other languages)

Figure 171: Comments per line of code (mean) of all tasks (Haskell vs. other languages)

Figure 172: Comments per line of code (mean) of all tasks (Haskell vs. other languages)

Figure 173: Comments per line of code (mean) of all tasks (Java vs. other languages)

Figure 174: Comments per line of code (mean) of all tasks (Java vs. other languages)

Figure 175: Comments per line of code (mean) of all tasks (Python vs. other languages)

Figure 176: Comments per line of code (mean) of all tasks (Python vs. other languages)

Figure 177: Comments per line of code (mean) of all tasks (all languages)

Figure 178: Size of binaries (min) of tasks compiling successfully (C vs. other languages)

Figure 179: Size of binaries (min) of tasks compiling successfully (C vs. other languages)

Figure 180: Size of binaries (min) of tasks compiling successfully (C# vs. other languages)

Figure 181: Size of binaries (min) of tasks compiling successfully (C# vs. other languages)

Figure 182: Size of binaries (min) of tasks compiling successfully (F# vs. other languages)

Figure 183: Size of binaries (min) of tasks compiling successfully (F# vs. other languages)

Figure 184: Size of binaries (min) of tasks compiling successfully (Go vs. other languages)

Figure 185: Size of binaries (min) of tasks compiling successfully (Go vs. other languages)

Figure 186: Size of binaries (min) of tasks compiling successfully (Haskell vs. other languages)

Figure 187: Size of binaries (min) of tasks compiling successfully (Haskell vs. other languages)

Figure 188: Size of binaries (min) of tasks compiling successfully (Java vs. other languages)

Figure 189: Size of binaries (min) of tasks compiling successfully (Java vs. other languages)

Figure 190: Size of binaries (min) of tasks compiling successfully (Python vs. other languages)

Figure 191: Size of binaries (min) of tasks compiling successfully (Python vs. other languages)

Figure 192: Size of binaries (min) of tasks compiling successfully (all languages)

Figure 193: Size of binaries (mean) of tasks compiling successfully (C vs. other languages)

Figure 194: Size of binaries (mean) of tasks compiling successfully (C vs. other languages)

Figure 195: Size of binaries (mean) of tasks compiling successfully (C# vs. other languages)

Figure 196: Size of binaries (mean) of tasks compiling successfully (C# vs. other languages)

Figure 197: Size of binaries (mean) of tasks compiling successfully (F# vs. other languages)

Figure 198: Size of binaries (mean) of tasks compiling successfully (F# vs. other languages)

Figure 199: Size of binaries (mean) of tasks compiling successfully (Go vs. other languages)

Figure 200: Size of binaries (mean) of tasks compiling successfully (Go vs. other languages)

Figure 201: Size of binaries (mean) of tasks compiling successfully (Haskell vs. other languages)

Figure 202: Size of binaries (mean) of tasks compiling successfully (Haskell vs. other languages)

Figure 203: Size of binaries (mean) of tasks compiling successfully (Java vs. other languages)

Figure 204: Size of binaries (mean) of tasks compiling successfully (Java vs. other languages)

Figure 205: Size of binaries (mean) of tasks compiling successfully (Python vs. other languages)

Figure 206: Size of binaries (mean) of tasks compiling successfully (Python vs. other languages)

Figure 207: Size of binaries (mean) of tasks compiling successfully (all languages)

Figure 208: Performance (min) of tasks running successfully (C vs. other languages)

Figure 209: Performance (min) of tasks running successfully (C vs. other languages)

Figure 210: Performance (min) of tasks running successfully (C# vs. other languages)

Figure 211: Performance (min) of tasks running successfully (C# vs. other languages)

Figure 212: Performance (min) of tasks running successfully (F# vs. other languages)

Figure 213: Performance (min) of tasks running successfully (F# vs. other languages)

Figure 214: Performance (min) of tasks running successfully (Go vs. other languages)

Figure 215: Performance (min) of tasks running successfully (Go vs. other languages)

Figure 216: Performance (min) of tasks running successfully (Haskell vs. other languages)

Figure 217: Performance (min) of tasks running successfully (Haskell vs. other languages)

Figure 218: Performance (min) of tasks running successfully (Java vs. other languages)

Figure 219: Performance (min) of tasks running successfully (Java vs. other languages)

Figure 220: Performance (min) of tasks running successfully (Python vs. other languages)

Figure 221: Performance (min) of tasks running successfully (Python vs. other languages)

Figure 222: Performance (min) of tasks running successfully (all languages)

Figure 223: Performance (mean) of tasks running successfully (C vs. other languages)

Figure 224: Performance (mean) of tasks running successfully (C vs. other languages)

Figure 225: Performance (mean) of tasks running successfully (C# vs. other languages)

Figure 226: Performance (mean) of tasks running successfully (C# vs. other languages)

Figure 227: Performance (mean) of tasks running successfully (F# vs. other languages)

Figure 228: Performance (mean) of tasks running successfully (F# vs. other languages)

Figure 229: Performance (mean) of tasks running successfully (Go vs. other languages)

Figure 230: Performance (mean) of tasks running successfully (Go vs. other languages)

Figure 231: Performance (mean) of tasks running successfully (Haskell vs. other languages)

Figure 232: Performance (mean) of tasks running successfully (Haskell vs. other languages)

Figure 233: Performance (mean) of tasks running successfully (Java vs. other languages)

Figure 234: Performance (mean) of tasks running successfully (Java vs. other languages)

Figure 235: Performance (mean) of tasks running successfully (Python vs. other languages)

Figure 236: Performance (mean) of tasks running successfully (Python vs. other languages)

Figure 237: Performance (mean) of tasks running successfully (all languages)

Figure 238: Scalability (min) of tasks running successfully (C vs. other languages)

Figure 239: Scalability (min) of tasks running successfully (C vs. other languages)

Figure 240: Scalability (min) of tasks running successfully (C# vs. other languages)

Figure 241: Scalability (min) of tasks running successfully (C# vs. other languages)

Figure 242: Scalability (min) of tasks running successfully (F# vs. other languages)

Figure 243: Scalability (min) of tasks running successfully (F# vs. other languages)

Figure 244: Scalability (min) of tasks running successfully (Go vs. other languages)

Figure 245: Scalability (min) of tasks running successfully (Go vs. other languages)

Figure 246: Scalability (min) of tasks running successfully (Haskell vs. other languages)

Figure 247: Scalability (min) of tasks running successfully (Haskell vs. other languages)

Figure 248: Scalability (min) of tasks running successfully (Java vs. other languages)

Figure 249: Scalability (min) of tasks running successfully (Java vs. other languages)

Figure 250: Scalability (min) of tasks running successfully (Python vs. other languages)

Figure 251: Scalability (min) of tasks running successfully (Python vs. other languages)

Figure 252: Scalability (min) of tasks running successfully (all languages)

Figure 253: Scalability (mean) of tasks running successfully (C vs. other languages)

Figure 254: Scalability (mean) of tasks running successfully (C vs. other languages)

Figure 255: Scalability (mean) of tasks running successfully (C# vs. other languages)

Figure 256: Scalability (mean) of tasks running successfully (C# vs. other languages)

Figure 257: Scalability (mean) of tasks running successfully (F# vs. other languages)

Figure 258: Scalability (mean) of tasks running successfully (F# vs. other languages)

Figure 259: Scalability (mean) of tasks running successfully (Go vs. other languages)

Figure 260: Scalability (mean) of tasks running successfully (Go vs. other languages)

Figure 261: Scalability (mean) of tasks running successfully (Haskell vs. other languages)

Figure 262: Scalability (mean) of tasks running successfully (Haskell vs. other languages)

Figure 263: Scalability (mean) of tasks running successfully (Java vs. other languages)

Figure 264: Scalability (mean) of tasks running successfully (Java vs. other languages)

Figure 265: Scalability (mean) of tasks running successfully (Python vs. other languages)

Figure 266: Scalability (mean) of tasks running successfully (Python vs. other languages)

Figure 267: Scalability (mean) of tasks running successfully (all languages)

Figure 268: Maximum RAM usage (min) of tasks running successfully (C vs. other languages)

Figure 269: Maximum RAM usage (min) of tasks running successfully (C vs. other languages)

Figure 270: Maximum RAM usage (min) of tasks running successfully (C# vs. other languages)

Figure 271: Maximum RAM usage (min) of tasks running successfully (C# vs. other languages)

Figure 272: Maximum RAM usage (min) of tasks running successfully (F# vs. other languages)

Figure 273: Maximum RAM usage (min) of tasks running successfully (F# vs. other languages)

Figure 274: Maximum RAM usage (min) of tasks running successfully (Go vs. other languages)

Figure 275: Maximum RAM usage (min) of tasks running successfully (Go vs. other languages)

Figure 276: Maximum RAM usage (min) of tasks running successfully (Haskell vs. other languages)

Figure 277: Maximum RAM usage (min) of tasks running successfully (Haskell vs. other languages)

Figure 278: Maximum RAM usage (min) of tasks running successfully (Java vs. other languages)

Figure 279: Maximum RAM usage (min) of tasks running successfully (Java vs. other languages)

Figure 280: Maximum RAM usage (min) of tasks running successfully (Python vs. other languages)

Figure 281: Maximum RAM usage (min) of tasks running successfully (Python vs. other languages)

Figure 282: Maximum RAM usage (min) of tasks running successfully (all languages)

Figure 283: Maximum RAM usage (mean) of tasks running successfully (C vs. other languages)

Figure 284: Maximum RAM usage (mean) of tasks running successfully (C vs. other languages)

Figure 285: Maximum RAM usage (mean) of tasks running successfully (C# vs. other languages)

Figure 286: Maximum RAM usage (mean) of tasks running successfully (C# vs. other languages)

Figure 287: Maximum RAM usage (mean) of tasks running successfully (F# vs. other languages)

Figure 288: Maximum RAM usage (mean) of tasks running successfully (F# vs. other languages)

Figure 289: Maximum RAM usage (mean) of tasks running successfully (Go vs. other languages)

Figure 290: Maximum RAM usage (mean) of tasks running successfully (Go vs. other languages)

Figure 291: Maximum RAM usage (mean) of tasks running successfully (Haskell vs. other languages)

Figure 292: Maximum RAM usage (mean) of tasks running successfully (Haskell vs. other languages)

Figure 293: Maximum RAM usage (mean) of tasks running successfully (Java vs. other languages)

Figure 294: Maximum RAM usage (mean) of tasks running successfully (Java vs. other languages)

Figure 295: Maximum RAM usage (mean) of tasks running successfully (Python vs. other languages)

Figure 296: Maximum RAM usage (mean) of tasks running successfully (Python vs. other languages)

Figure 297: Maximum RAM usage (mean) of tasks running successfully (all languages)

Figure 298: Page faults (min) of tasks running successfully (C vs. other languages)

Figure 299: Page faults (min) of tasks running successfully (C vs. other languages)

Figure 300: Page faults (min) of tasks running successfully (C# vs. other languages)

Figure 301: Page faults (min) of tasks running successfully (C# vs. other languages)

Figure 302: Page faults (min) of tasks running successfully (F# vs. other languages)

Figure 303: Page faults (min) of tasks running successfully (F# vs. other languages)

Figure 304: Page faults (min) of tasks running successfully (Go vs. other languages)

Figure 305: Page faults (min) of tasks running successfully (Go vs. other languages)

Figure 306: Page faults (min) of tasks running successfully (Haskell vs. other languages)

Figure 307: Page faults (min) of tasks running successfully (Haskell vs. other languages)

Figure 308: Page faults (min) of tasks running successfully (Java vs. other languages)

Figure 309: Page faults (min) of tasks running successfully (Java vs. other languages)

Figure 310: Page faults (min) of tasks running successfully (Python vs. other languages)

Figure 311: Page faults (min) of tasks running successfully (Python vs. other languages)

Figure 312: Page faults (min) of tasks running successfully (all languages)

Figure 313: Page faults (mean) of tasks running successfully (C vs. other languages)

Figure 314: Page faults (mean) of tasks running successfully (C vs. other languages)

Figure 315: Page faults (mean) of tasks running successfully (C# vs. other languages)

Figure 316: Page faults (mean) of tasks running successfully (C# vs. other languages)

Figure 317: Page faults (mean) of tasks running successfully (F# vs. other languages)

Figure 318: Page faults (mean) of tasks running successfully (F# vs. other languages)

Figure 319: Page faults (mean) of tasks running successfully (Go vs. other languages)

Figure 320: Page faults (mean) of tasks running successfully (Go vs. other languages)

Figure 321: Page faults (mean) of tasks running successfully (Haskell vs. other languages)

Figure 322: Page faults (mean) of tasks running successfully (Haskell vs. other languages)

Figure 323: Page faults (mean) of tasks running successfully (Java vs. other languages)

Figure 324: Page faults (mean) of tasks running successfully (Java vs. other languages)

Figure 325: Page faults (mean) of tasks running successfully (Python vs. other languages)

Figure 326: Page faults (mean) of tasks running successfully (Python vs. other languages)

Figure 327: Page faults (mean) of tasks running successfully (all languages)

Figure 328: Timeout analysis of scalability tasks (C vs. other languages)

Figure 329: Timeout analysis of scalability tasks (C vs. other languages)

Figure 330: Timeout analysis of scalability tasks (C# vs. other languages)

Figure 331: Timeout analysis of scalability tasks (C# vs. other languages)

Figure 332: Timeout analysis of scalability tasks (F# vs. other languages)

Figure 333: Timeout analysis of scalability tasks (F# vs. other languages)

Figure 334: Timeout analysis of scalability tasks (Go vs. other languages)

Figure 335: Timeout analysis of scalability tasks (Go vs. other languages)

Figure 336: Timeout analysis of scalability tasks (Haskell vs. other languages)

Figure 337: Timeout analysis of scalability tasks (Haskell vs. other languages)

Figure 338: Timeout analysis of scalability tasks (Java vs. other languages)

Figure 339: Timeout analysis of scalability tasks (Java vs. other languages)

Figure 340: Timeout analysis of scalability tasks (Python vs. other languages)

Figure 341: Timeout analysis of scalability tasks (Python vs. other languages)

Figure 342: Timeout analysis of scalability tasks (all languages)

Figure 343: Number of solutions per task (C vs. other languages)

Figure 344: Number of solutions per task (C vs. other languages)

Figure 345: Number of solutions per task (C# vs. other languages)

Figure 346: Number of solutions per task (C# vs. other languages)

Figure 347: Number of solutions per task (F# vs. other languages)

Figure 348: Number of solutions per task (F# vs. other languages)

Figure 349: Number of solutions per task (Go vs. other languages)

Figure 350: Number of solutions per task (Go vs. other languages)

Figure 351: Number of solutions per task (Haskell vs. other languages)

Figure 352: Number of solutions per task (Haskell vs. other languages)

Figure 353: Number of solutions per task (Java vs. other languages)

Figure 354: Number of solutions per task (Java vs. other languages)

Figure 355: Number of solutions per task (Python vs. other languages)

Figure 356: Number of solutions per task (Python vs. other languages)

Figure 357: Number of solutions per task (all languages)