Static v. dynamic languages

There are some pretty strong statements about types floating around out there. The claims range from the oft-repeated phrase that when you get the types to line up, everything just works, to “not relying on type safety is unethical (if you have an SLA)”[1], "It boils down to cost vs benefit, actual studies, and mathematical axioms, not aesthetics or feelings", and I think programmers who doubt that type systems help are basically the tech equivalent of an anti-vaxxer. The first and last of these statements are from "types" thought leaders who are widely quoted. There are probably plenty of strong claims about dynamic languages that I'd be skeptical of if I heard them, but I'm not in the right communities to hear the stronger claims about dynamically typed languages. Either way, it's rare to see people cite actual evidence.

Let's take a look at the empirical evidence that backs up these claims.

Click here if you just want to see the summary without having to wade through all the studies. The summary of the summary is that most studies find very small effects, if any. However, the studies probably don't cover contexts you're actually interested in. If you want the gory details, here's each study, with its abstract, and a short blurb about the study.

A Large Scale Study of Programming Languages and Code Quality in Github; Ray, B; Posnett, D; Filkov, V; Devanbu, P

Abstract

> What is the effect of programming languages on software quality? This question has been a topic of much debate for a very long time. In this study, we gather a very large data set from GitHub (729 projects, 80 Million SLOC, 29,000 authors, 1.5 million commits, in 17 languages) in an attempt to shed some empirical light on this question. This reasonably large sample size allows us to use a mixed-methods approach, combining multiple regression modeling with visualization and text analytics, to study the effect of language features such as static v.s. dynamic typing, strong v.s. weak typing on software quality. By triangulating findings from different methods, and controlling for confounding effects such as team size, project size, and project history, we report that language design does have a significant, but modest effect on software quality. Most notably, it does appear that strong typing is modestly better than weak typing, and among functional languages, static typing is also somewhat better than dynamic typing. We also find that functional languages are somewhat better than procedural languages. It is worth noting that these modest effects arising from language design are overwhelmingly dominated by the process factors such as project size, team size, and commit size. However, we hasten to caution the reader that even these modest effects might quite possibly be due to other, intangible process factors, e.g., the preference of certain personality types for functional, static and strongly typed languages.

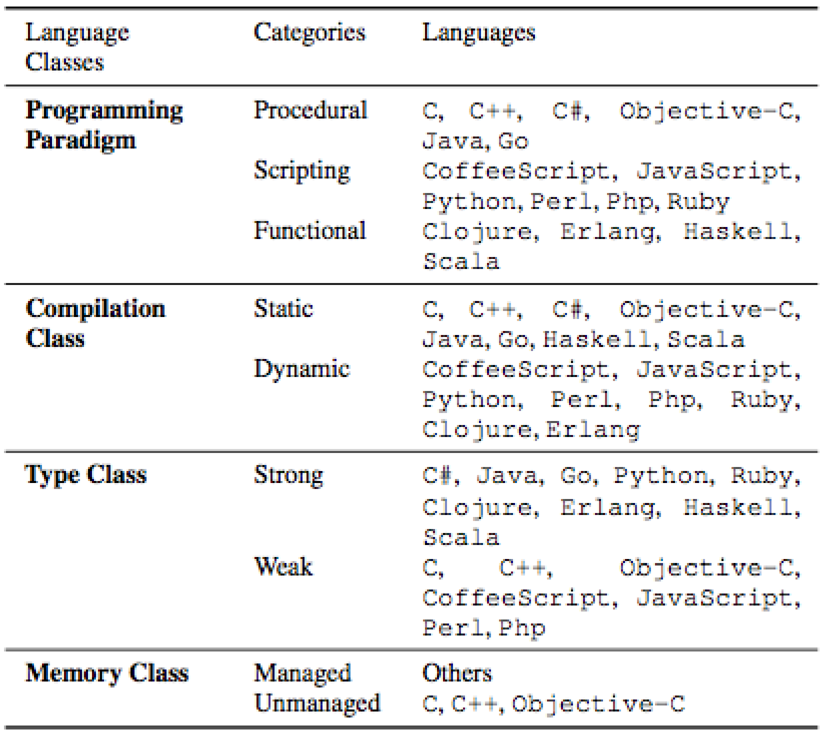
Summary

The authors looked at the 50 most starred repos on github for each of the 20 most popular languages plus TypeScript (minus CSS, shell, and vim). For each of these projects, they looked at the languages used. The text in the body of the study doesn't support the strong claims made in the abstract. Additionally, the study appears to use a fundamentally flawed methodology that's not capable of revealing much information. Even if the methodology were sound, the study uses bogus data and has what Pinker calls the igon value problem.

As Gary Bernhardt points out, the authors of the study seem to confuse memory safety and implicit coercion and make other strange statements, such as

> Advocates of dynamic typing may argue that rather than spend a lot of time correcting annoying static type errors arising from sound, conservative static type checking algorithms in compilers, it’s better to rely on strong dynamic typing to catch errors as and when they arise.

The study uses the following language classification scheme

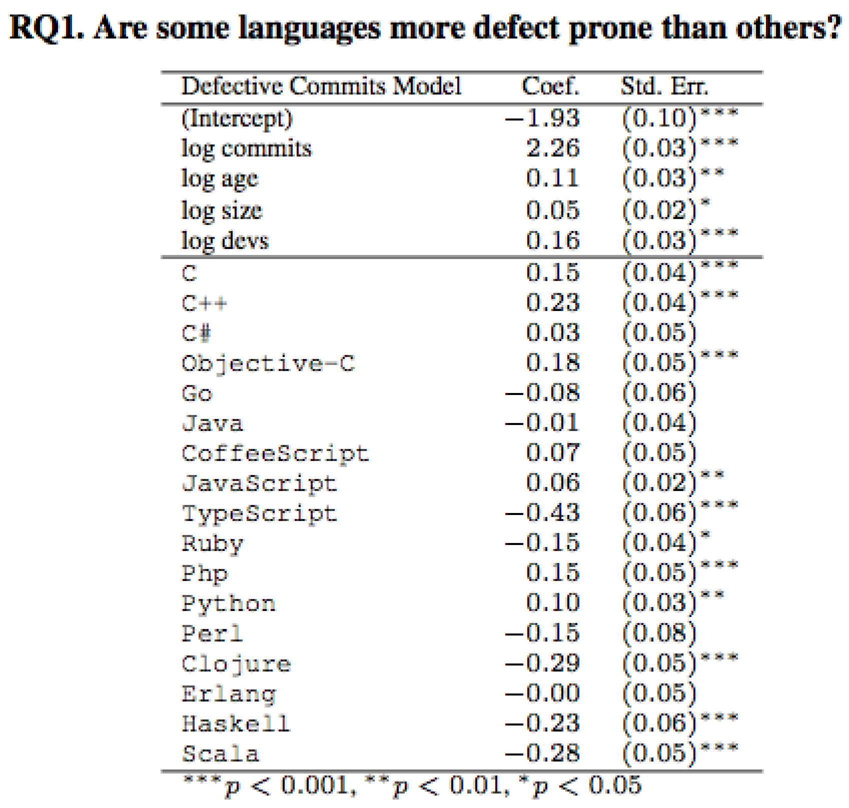


Language Classes	Categories	Languages
Programming Paradigm	Procedural	C, C++, C#, Objective-C, Java, Go
	Scripting	CoffeeScript, JavaScript, Python, Perl, Php, Ruby
	Functional	Clojure, Erlang, Haskell, Scala
Compilation Class	Static	C, C++, C#, Objective-C, Java, Go, Haskell, Scala
	Dynamic	CoffeeScript, JavaScript, Python, Perl, Php, Ruby, Clojure, Erlang
Type Class	Strong	C#, Java, Go, Python, Ruby, Clojure, Erlang, Haskell, Scala
	Weak	C, C++, Objective-C, CoffeeScript, JavaScript, Perl, Php
Memory Class	Managed	Others
	Unmanaged	C, C++, Objective-C

These classifications seem arbitrary and many people would disagree with some of these classifications. Since the results are based on aggregating results with respect to these categories, and the authors have chosen arbitrary classifications, this already makes the aggragated results suspect since they have a number of degrees of freedom here and they've made some odd choicses.

In order to get the language level results, the authors looked at commit/PR logs to determine how many bugs there were for each language used. As far as I can tell, open issues with no associated fix don't count towards the bug count. Only commits that are detected by their keyword search technique were counted. With this methodology, the number of bugs found will depend at least as strongly on the bug reporting culture as it does on the actual number of bugs found.

After determining the number of bugs, the authors ran a regression, controlling for project age, number of developers, number of commits, and lines of code.



RQ1. Are some languages more defect prone than others?

Defective Commits Model	Coef.	Std. Err.
(Intercept)	−1.93	(0.10)***
log commits	2.26	(0.03)***
log age	0.11	(0.03)**
log size	0.05	(0.02)*
log devs	0.16	(0.03)***
C	0.15	(0.04)***
C++	0.23	(0.04)***
C#	0.03	(0.05)
Objective-C	0.18	(0.05)***
Go	−0.08	(0.06)
Java	−0.01	(0.04)
CoffeeScript	0.07	(0.05)
JavaScript	0.06	(0.02)**
TypeScript	−0.43	(0.06)***
Ruby	−0.15	(0.04)*
Php	0.15	(0.05)***
Python	0.10	(0.03)**
Perl	−0.15	(0.08)
Clojure	−0.29	(0.05)***
Erlang	−0.00	(0.05)
Haskell	−0.23	(0.06)***
Scala	−0.28	(0.05)***

***$p < 0.001$, **$p < 0.01$, *$p < 0.05$

There are enough odd correlations here that, even if the methodology wasn't known to be flawed, I'd be skeptical that authors have captured a causal relationship. If you don't find it odd that Perl and Ruby are as reliable as each other and significantly more reliable than Erlang and Java (which are also equally reliable), which are significantly more reliable than Python, PHP, and C (which are similarly reliable), and that TypeScript is the safest language surveyed, then maybe this passes the sniff test for you, but even without reading further, this looks suspicious.

For example, Erlang and Go are rated as having a lot of concurrency bugs, whereas Perl and CoffeeScript are rated as having few concurrency bugs. Is it more plausible that Perl and CoffeeScript are better at concurrency than Erlang and Go or that people tend to use Erlang and Go more when they need concurrency? The authors note that Go might have a lot of concurrency bugs because there's a good tool to detect concurrency bugs in Go, but they don't explore reasons for most of the odd intermediate results.

As for TypeScript, the three projects they list as example TypeScript projects (bitcoin, litecoin, and qBittorrent) are C++ projects. So the intermediate result appears to not be that TypeScript is reliable, but that projects mis-identified as TypeScript are reliable. Those projects are reliable because Qt translation files are identified as TypeScript and it turns out that, per line of code, giant dumps of config files from another project don't cause a lot of bugs. It's like saying that a project has few bugs per line of code because it has a giant README. This is the most blatant classification error, but it's far from the only one. Since this study uses Github's notoriously inaccurate code classification system to classify repos, it is, at best, a series of correlations with factors that are themselves only loosely correlated with actual language usage.

There's more analysis, but much of it is based on aggregating the table above into categories based on language type. Since I'm skeptical of these results, I'm at least as skeptical of any results based on aggregating these results. This section barely even scratches the surface of this study. Even with just a light skim, we see multiple serious flaws, any one of which would invalidate the results, plus numerous igon value problems.

A controlled experiment to assess the benefits of procedure argument type checking, Prechelt, L.; Tichy, W.F.

Abstract

Type checking is considered an important mechanism for detecting programming errors, especially interface errors. This report describes an experiment to assess the defect-detection capabilities of static, intermodule type checking.

The experiment uses ANSI C and Kernighan & Ritchie (K&R) C. The relevant difference is that the ANSI C compiler checks module interfaces (i.e., the parameter lists calls to external functions), whereas K&R C does not. The experiment employs a counterbalanced design in which each of the 40 subjects, most of them CS PhD students, writes two nontrivial programs that interface with a complex library (Motif). Each subject writes one program in ANSI C and one in K&R C. The input to each compiler run is saved and manually analyzed for defects.

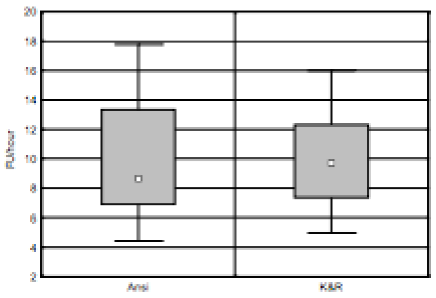
Results indicate that delivered ANSI C programs contain significantly fewer interface defects than delivered K&R C programs. Furthermore, after subjects have gained some familiarity with the interface they are using, ANSI C programmers remove defects faster and are more productive (measured in both delivery time and functionality implemented)

Summary

The “nontrivial” tasks are the inversion of a 2x2 matrix (with GUI) and a file “browser” menu that has two options, select file and display file. Docs for motif were provided, but example code was deliberately left out.

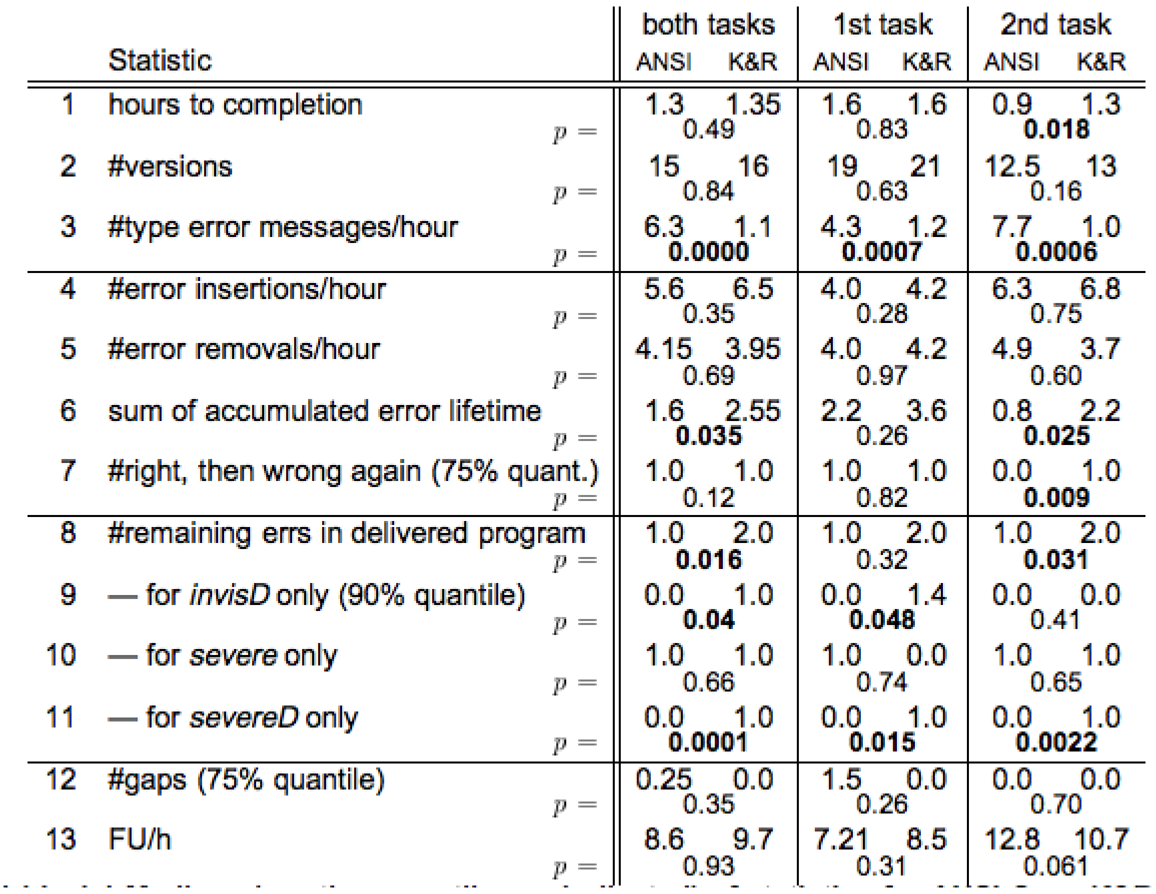
There are 34 subjects. Each subjects solves one problem with the K&R C compiler (which doesn't typecheck arguments) and one with the ANSI C compiler (which does).

The authors note that the distribution of results is non-normal, with highly skewed outliers, but they present their results as box plots, which makes it impossible to see the distribution. They do some statistical significance tests on various measures, and find no difference in time to completion on the first task, a significant difference on the second task, but no difference when the tasks are pooled.



In terms of how the bugs are introduced during the programming process, they do a significance test against the median of one measure of defects (which finds a significant difference in the first task but not the second), and a significance test against the 75%-quantile of another measure (which finds a significant difference in the second task but not the first).

In terms of how many and what sort of bugs are in the final program, they define a variety of measures and find that some differences on the measures are statistically significant and some aren't. In the table below, bolded values indicate statistically significant differences.



	Statistic	both tasks		1st task		2nd task	
		ANSI	K&R	ANSI	K&R	ANSI	K&R
1	hours to completion	1.3	1.35	1.6	1.6	0.9	1.3
	$p =$	0.49		0.83		**0.018**	
2	#versions	15	16	19	21	12.5	13
	$p =$	0.84		0.63		0.16	
3	#type error messages/hour	6.3	1.1	4.3	1.2	7.7	1.0
	$p =$	**0.0000**		**0.0007**		**0.0006**	
4	#error insertions/hour	5.6	6.5	4.0	4.2	6.3	6.8
	$p =$	0.35		0.28		0.75	
5	#error removals/hour	4.15	3.95	4.0	4.2	4.9	3.7
	$p =$	0.69		0.97		0.60	
6	sum of accumulated error lifetime	1.6	2.55	2.2	3.6	0.8	2.2
	$p =$	**0.035**		0.26		**0.025**	
7	#right, then wrong again (75% quant.)	1.0	1.0	1.0	1.0	0.0	1.0
	$p =$	0.12		0.82		**0.009**	
8	#remaining errs in delivered program	1.0	2.0	1.0	2.0	1.0	2.0
	$p =$	**0.016**		0.32		**0.031**	
9	— for *invisD* only (90% quantile)	0.0	1.0	0.0	1.4	0.0	0.0
	$p =$	**0.04**		**0.048**		0.41	
10	— for *severe* only	1.0	1.0	1.0	0.0	1.0	1.0
	$p =$	0.66		0.74		0.65	
11	— for *severeD* only	0.0	1.0	0.0	1.0	0.0	1.0
	$p =$	**0.0001**		**0.015**		**0.0022**	
12	#gaps (75% quantile)	0.25	0.0	1.5	0.0	0.0	0.0
	$p =$	0.35		0.26		0.70	
13	FU/h	8.6	9.7	7.21	8.5	12.8	10.7
	$p =$	0.93		0.31		0.061	

Note that here, first task refers to whichever task the subject happened to perform first, which is randomized, which makes the results seem rather arbitrary. Furthermore, the numbers they compare are medians (except where indicated otherwise), which also seems arbitrary.

Despite the strong statement in the abstract, I'm not convinced this study presents strong evidence for anything in particular. They have multiple comparisons, many of which seem arbitrary, and find that some of them are significant. They also find that many of their criteria don't have significant differences. Furthermore, they don't mention whether or not they tested any other arbitrary criteria. If they did, the results are much weaker than they look, and they already don't look strong.

My interpretation of this is that, if there is an effect, the effect is dwarfed by the difference between programmers, and it's not clear whether there's any real effect at all.

An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl, Prechelt, L.

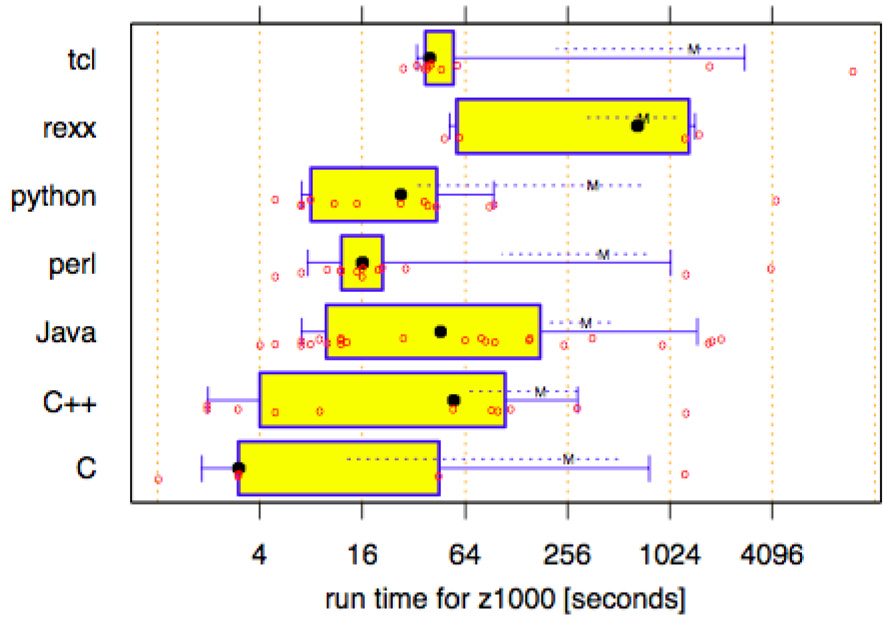
Abstract

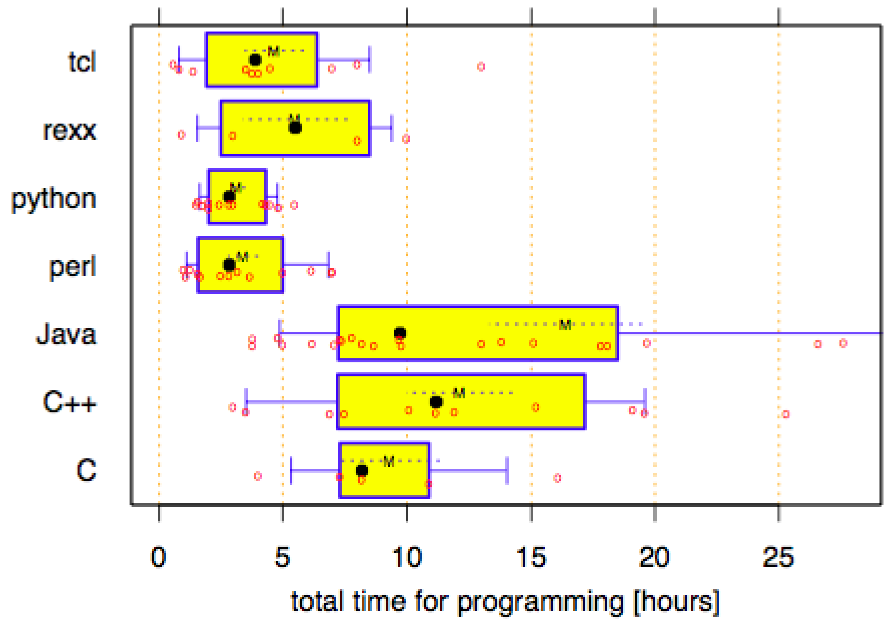
80 implementations of the same set of requirements are compared for several properties, such as run time, memory consumption, source text length, comment density, program structure, reliability, and the amount of effort required for writing them. The results indicate that, for the given programming problem, which regards string manipulation and search in a dictionary, “scripting languages” (Perl, Python, Rexx, Tcl) are more productive than “conventional languages” (C, C++, Java). In terms of run time and memory consumption, they often turn out better than Java and not much worse than C or C++. In general, the differences between languages tend to be smaller than the typical differences due to different programmers within the same language.

Summary

The task was to read in a list of phone numbers and return a list of words that those phone numbers could be converted to, using the letters on a phone keypad.

This study was done in two phases. There was a controlled study for the C/C++/Java group, and a self-timed implementation for the Perl/Python/Rexx/Tcl group. The former group consisted of students while the latter group consisted of respondents from a newsgroup. The former group received more criteria they should consider during implementation, and had to implement the program when they received the problem description, whereas some people in the latter group read the problem description days or weeks before implementation.





If you take the results at face value, it looks like the class of language used imposes a lower bound on both implementation time and execution time, but that the variance between programmers is much larger than the variance between languages.

However, since the scripting language group had significantly different (and easier) environment than the C-like language group, it's hard to say how much of the measured difference in implementation time is from flaws in the experimental design and how much is real.

Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study Kleinschmager, S.; Hanenberg, S.; Robbes, R.; Tanter, E.; Stefik, A.

Abstract

> Static type systems play an essential role in contemporary programming languages. Despite their importance, whether static type systems influence human software development capabilities remains an open question. One frequently mentioned argument for static type systems is that they improve the maintainability of software systems - an often used claim for which there is no empirical evidence. This paper describes an experiment which tests whether static type systems improve the maintainability of software systems. The results show rigorous empirical evidence that static type are indeed beneficial to these activities, except for fixing semantic errors.

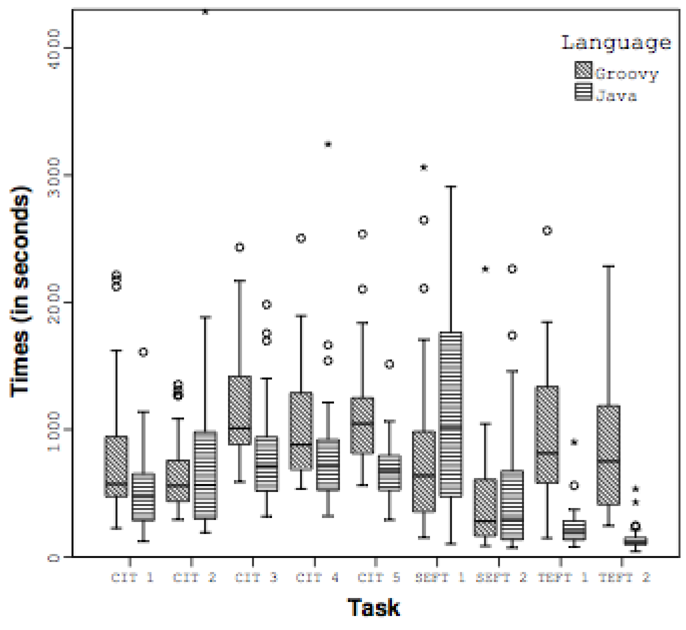
Summary

While the abstract talks about general classes of languages, the study uses Java and Groovy.

Subjects were given classes in which they had to either fix errors in existing code or fill out stub methods. Static classes for Java, dynamic classes for Groovy. In cases of type errors (and their respective no method errors), developers solved the problem faster in Java. For semantic errors, there was no difference.

The study used a within-subject design, with randomized task order over 33 subjects.

A notable limitation is that the study avoided using "complicated control structures", such as loops and recursion, because those increase variance in time-to-solve. As a result, all of the bugs are trivial bugs. This can be seen in the median time to solve the tasks, which are in the hundreds of seconds. Tasks can include multiple bugs, so the time per bug is quite low.



This paper mentions that its results contradict some prior results, and one of the possible causes they give is that their tasks are more complex than the tasks from those other papers. The fact that the tasks in this paper don't involve using loops and recursion because they're too complicated, should give you an idea of the complexity of the tasks involved in most of these papers.

Other limitations in this experiment were that the variables were artificially named such that there was no type information encoded in any of the names, that there were no comments, and that there was zero documentation on the APIs provided. That's an unusually hostile environment to find bugs in, and it's not clear how the results generalize if any form of documentation is provided.

Additionally, even though the authors specifically picked trivial tasks in order to minimize the variance between programmers, the variance between programmers was still much greater than the variance between languages in all but two tasks. Those two tasks were both cases of a simple type error causing a run-time exception that wasn't near the type error.

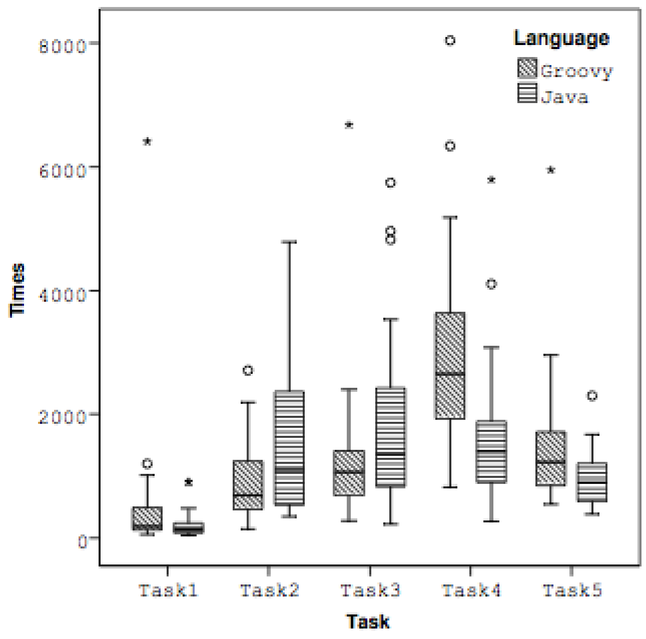
Static type systems (sometimes) have a positive impact on the usability of undocumented software; Mayer, C.; Hanenberg, S.; Robbes, R.; Tanter, E.; Stefik, A.

Abstract

Static and dynamic type systems (as well as more recently gradual type systems) are an important research topic in programming language design. Although the study of such systems plays a major role in research, relatively little is known about the impact of type systems on software development. Perhaps one of the more common arguments for static type systems is that they require developers to annotate their code with type names, which is thus claimed to improve the documentation of software. In contrast, one common argument against static type systems is that they decrease flexibility, which may make them harder to use. While positions such as these, both for and against static type systems, have been documented in the literature, there is little rigorous empirical evidence for or against either position. In this paper, we introduce a controlled experiment where 27 subjects performed programming tasks on an undocumented API with a static type system (which required type annotations) as well as a dynamic type system (which does not). Our results show that for some types of tasks, programmers were afforded faster task completion times using a static type system, while for others, the opposite held. In this work, we document the empirical evidence that led us to this conclusion and conduct an exploratory study to try and theorize why.

Summary

The experimental setup is very similar to the previous Hanenberg paper, so I'll just describe the main difference, which is that subjects used either Java, or a restricted subset of Groovy that was equivalent to dynamically typed Java. Subjects were students who had previous experience in Java, but not Groovy, giving some advantage for the Java tasks.



Task 1 was a trivial warm-up task. The authors note that it's possible that Java is superior on task 1 because the subjects had prior experience in Java. The authors speculate that, in general, Java is superior to untyped Java for more complex tasks, but they make it clear that they're just speculating and don't have enough data to conclusively support that conclusion.

How Do API Documentation and Static Typing Affect API Usability? Endrikat, S.; Hanenberg, S.; Robbes, Romain; Stefik, A.

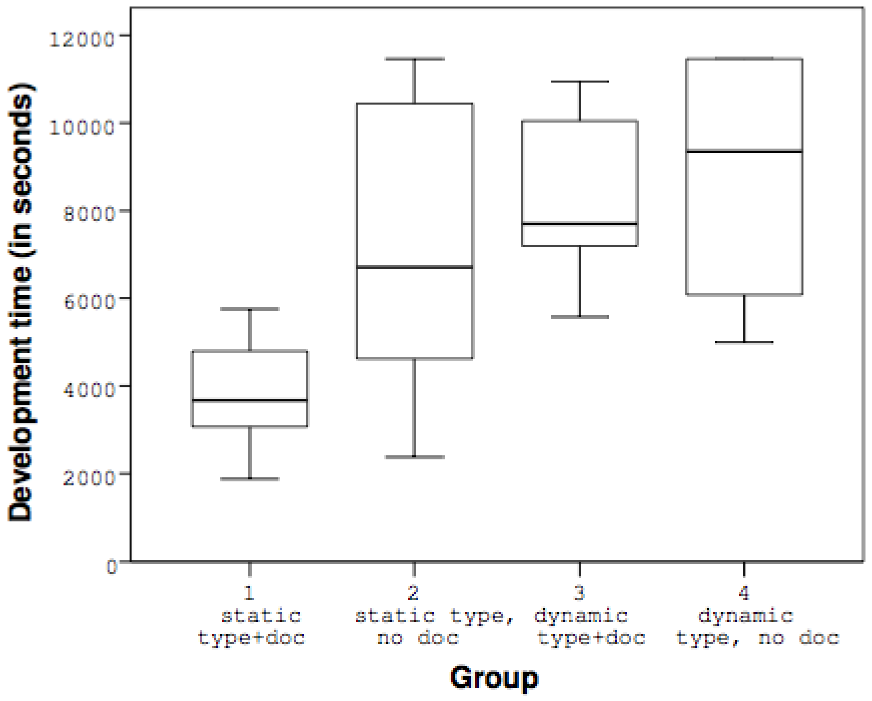
Abstract

> When developers use Application Programming Interfaces (APIs), they often rely on documentation to assist their tasks. In previous studies, we reported evidence indicating that static type systems acted as a form of implicit documentation, benefiting developer productivity. Such implicit documentation is easier to maintain, given it is enforced by the compiler, but previous experiments tested users without any explicit documentation. In this paper, we report on a controlled experiment and an exploratory study comparing the impact of using documentation and a static or dynamic type system on a development task. Results of our study both confirm previous findings and show that the benefits of static typing are strengthened with explicit documentation, but that this was not as strongly felt with dynamically typed languages.

There's an earlier study in this series with the following abstract:

> In the discussion about the usefulness of static or dynamic type systems there is often the statement that static type systems improve the documentation of software. In the meantime there exists even some empirical evidence for this statement. One of the possible explanations for this positive influence is that the static type system of programming languages such as Java require developers to write down the type names, i.e. lexical representations which potentially help developers. Because of that there is a plausible hypothesis that the main benefit comes from the type names and not from the static type checks that are based on these names. In order to argue for or against static type systems it is desirable to check this plausible hypothesis in an experimental way. This paper describes an experiment with 20 participants that has been performed in order to check whether developers using an unknown API already benefit (in terms of development time) from the pure syntactical representation of type names without static type checking. The result of the study is that developers do benefit from the type names in an API's source code. But already a single wrong type name has a measurable significant negative impact on the development time in comparison to APIs without type names.

The languages used were Java and Dart. The university running the tests teaches in Java, so subjects had prior experience in Java. The task was one “where participants use the API in a way that objects need to be configured and passed to the API”, which was chosen because the authors thought that both types and documentation should have some effect. “The challenge for developers is to locate all the API elements necessary to properly configure [an] object”. The documentation was free-form text plus examples.



Taken at face value, it looks like types+documentation is a lot better than having one or the other, or neither. But since the subjects were students at a school that used Java, it's not clear how much of the effect is from familiarity with the language and how much is from the language. Moreover, the task was a single task that was chosen specifically because it was the kind of task where both types and documentation were expected to matter.

An Experiment About Static and Dynamic Type Systems; Hanenberg, S.

Abstract

> Although static type systems are an essential part in teaching and research in software engineering and computer science, there is hardly any knowledge about what the impact of static type systems on the development time or the resulting quality for a piece of software is. On the one hand there are authors that state that static type systems decrease an application's complexity and hence its development time (which means that the quality must be improved since developers have more time left in their projects). On the other hand there are authors that argue that static type systems increase development time (and hence decrease the code quality) since they restrict developers to express themselves in a desired way. This paper presents an empirical study with 49 subjects that studies the impact of a static type system for the development of a parser over 27 hours working time. In the experiments the existence of the static type system has neither a positive nor a negative impact on an application's development time (under the conditions of the experiment).

Summary

This is another Hanenberg study with a basically sound experimental design, so I won't go into details about the design. Some unique parts are that, in order to control for familiarity and other things that are difficult to control for with existing languages, the author created two custom languages for this study.

The author says that the language has similarities to Smalltalk, Ruby, and Java, and that the language is a class-based OO language with single implementation inheritance and late binding.

The students had 16 hours of training in the new language before starting. The author argues that this was sufficient because “the language, its API as well as its IDE was kept very simple”. An additional 2 hours was spent to explain the type system for the static types group.

There were two tasks, a “small” one (implementing a scanner) and a “large” one (implementing a parser). The author found a statistically significant difference in time to complete the small task (the dynamic language was faster) and no difference in the time to complete the large task.

There are a number of reasons this result may not be generalizable. The author is aware of them and there's a long section on ways this study doesn't generalize as well as a good discussion on threats to validity.

Work In Progress: an Empirical Study of Static Typing in Ruby; Daly, M; Sazawal, V; Foster, J.

Abstract

> In this paper, we present an empirical pilot study of four skilled programmers as they develop programs in Ruby, a popular, dynamically typed, object-oriented scripting language. Our study compares programmer behavior under the standard Ruby interpreter versus using Diamondback Ruby (DRuby), which adds static type inference to Ruby. The aim of our study is to understand whether DRuby's static typing is beneficial to programmers. We found that DRuby's warnings rarely provided information about potential errors not already evident from Ruby's own error messages or from presumed prior knowledge. We hypothesize that programmers have ways of reasoning about types that compensate for the lack of static type information, possibly limiting DRuby's usefulness when used on small programs.

Summary

Subjects came from a local Ruby user's group. Subjects implemented a simplified Sudoku solver and a maze solver. DRuby was randomly selected for one of the two problems for each subject. There were four subjects, but the authors changed the protocol after the first subject. Only three subjects had the same setup.

The authors find no benefit to having types. This is one of the studies that the first Hanenberg study mentions as a work their findings contradict. That first paper claimed that it was because their tasks were more complex, but it seems to me that this paper has a more complex task. One possible reason they found contradictory results is that the effect size is small. Another is that the specific type systems used matter, and that a DRuby v. Ruby study doesn't generalize to Java v. Groovy. Another is that the previous study attempted to remove anything hinting at type information from the dynamic implementation, including names that indicate types and API documentation. The participants of this study mention that they get a lot of type information from API docs, and the authors note that the participants encode type information in their method names.

This study was presented in a case study format, with selected comments from the participants and an analysis of their comments. The authors note that participants regularly think about types, and check types, even when programming in a dynamic language.

Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity; Hudak, P; Jones, M.

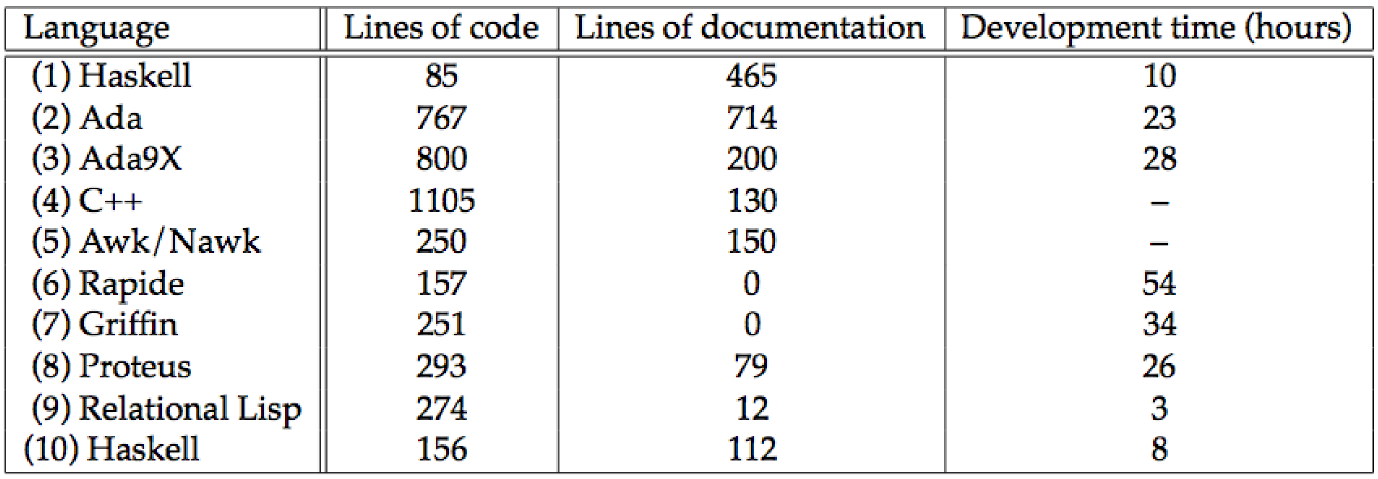
Abstract

We describe the results of an experiment in which several conventional programming languages, together with the functional language Haskell, were used to prototype a Naval Surface Warfare Center (NSWC) requirement for a Geometric Region Server. The resulting programs and development metrics were reviewed by a committee chosen by the Navy. The results indicate that the Haskell prototype took significantly less time to develop and was considerably more concise and easier to understand than the corresponding prototypes written in several different imperative languages, including Ada and C++.

Summary

Subjects were given an informal text description for the requirements of a geo server. The requirements were behavior oriented and didn't mention performance. The subjects were “expert” programmers in the languages they used. They were asked to implement a prototype and track metrics such as dev time, lines of code, and docs. Metrics were all self reported, and no guidelines were given as to how they should be measured, so metrics varied between subjects. Also, some, but not all, subjects attended a meeting where additional information was given on the assignment.

Due to the time-frame and funding requirements, the requirements for the server were extremely simple; the median implementation was a couple hundred lines of code. Furthermore, the panel that reviewed the solutions didn't have time to evaluate or run the code; they based their findings on the written reports and oral presentations of the subjects.



Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

This study hints at a very interesting result, but considering all of its limitations, the fact that each language (except Haskell) was only tested once, and that other studies show much larger intra-group variance than inter-group variance, it's hard to conclude much from this study alone.

Unit testing isn't enough. You need static typing too; Farrer, E

Abstract

> Unit testing and static type checking are tools for ensuring defect free software. Unit testing is the practice of writing code to test individual units of a piece of software. By validating each unit of software, defects can be discovered during development. Static type checking is performed by a type checker that automatically validates the correct typing of expressions and statements at compile time. By validating correct typing, many defects can be discovered during development. Static typing also limits the expressiveness of a programming language in that it will reject some programs which are ill-typed, but which are free of defects.
>
> Many proponents of unit testing claim that static type checking is an insufficient mechanism for ensuring defect free software; and therefore, unit testing is still required if static type checking is utilized. They also assert that once unit testing is utilized, static type checking is no longer needed for defect detection, and so it should be eliminated.
>
> The goal of this research is to explore whether unit testing does in fact obviate static type checking in real world examples of unit tested software.

Summary

The author took four Python programs and translated them to Haskell. Haskell's type system found some bugs. This is the first study we've looked at that involves something larger than a toy program, and it's also the only study that looks at a type system that's more expressive than Java's type system. The programs were the NMEA Toolkit (9 bugs), MIDITUL (2 bugs), GrapeFruit (0 bugs), and PyFontInfo (6 bugs).

As far as I can tell, there isn't an analysis of the severity of the bugs. The programs were 2324, 2253, 2390, and 609 lines long, respectively, so the bugs found / LOC were 17 / 7576 = 1 / 446. For reference, in Code Complete, Steve McConnell estimates that 15-50 bugs per 1kLOC is normal. If you believe that estimate applies to this codebase, you'd expect that this technique caught between 4% and 15% of the bugs in this code. There's no particular reason to believe the estimate should apply, but we can keep this number in mind as a reference in order to compare to a similarly generated number from another study that we'll get to later.

The author does some analysis on how hard it would have been to find the bugs through testing, but only considers line coverage directed unit testing; the author comments that bugs might have have been caught by unit testing if they could be missed with 100% line coverage. This seems artificially weak -- it's generally well accepted that line coverage is a very weak notion of coverage and that testing merely to get high line coverage isn't sufficient. In fact, it is generally considered insufficient to even test merely to get high path coverage, which is a much stronger notion of coverage than line coverage.

Gradual Typing of Erlang Programs: A Wrangler Experience; Sagonas, K; Luna, D

Abstract

> Currently most Erlang programs contain no or very little type information. This sometimes makes them unreliable, hard to use, and difficult to understand and maintain. In this paper we describe our experiences from using static analysis tools to gradually add type information to a medium sized Erlang application that we did not write ourselves: the code base of Wrangler. We carefully document the approach we followed, the exact steps we took, and discuss possible difficulties that one is expected to deal with and the effort which is required in the process. We also show the type of software defects that are typically brought forward, the opportunities for code refactoring and improvement, and the expected benefits from embarking in such a project. We have chosen Wrangler for our experiment because the process is better explained on a code base which is small enough so that the interested reader can retrace its steps, yet large enough to make the experiment quite challenging and the experiences worth writing about. However, we have also done something similar on large parts of Erlang/OTP. The result can partly be seen in the source code of Erlang/OTP R12B-3.

Summary

This is somewhat similar to the study in “Unit testing isn't enough”, except that the authors of this study created a static analysis tool instead of translating the program into another language. The authors note that they spent about half an hour finding and fixing bugs after running their tool. They also point out some bugs that would be difficult to find by testing. They explicitly state “what's interesting in our approach is that all these are achieved without imposing any (restrictive) static type system in the language.” The authors have a follow-on paper, “Static Detection of Race Conditions in Erlang”, which extends the approach.

The list of papers that find bugs using static analysis without explicitly adding types is too long to list. This is just one typical example.

0install: Replacing Python; Leonard, T., pt2, pt3

Abstract

No abstract because this is a series of blog posts.

Summary

This compares ATS, C#, Go, Haskell, OCaml, Python and Rust. The author assigns scores to various criteria, but it's really a qualitative comparison. But it's interesting reading because it seriously considers the effect of language on a non-trivial codebase (30kLOC).

The author implemented parts of 0install in various languages and then eventually decided on Ocaml and ported the entire thing to Ocaml. There are some great comments about why the author chose Ocaml and what the author gained by using Ocaml over Python.

The Unexpected Results From A Hardware Design Contest; Cooley, J

Abstract

No abstract because it's a usenet posting

Summary

Subjects were given 90 minutes to create a small chunk of hardware, a synchronous loadable 9-bit increment-by-3 decrement-by-5 up/down counter that generated even parity, carry and borrow, with the goal of optimizing for cycle time of the synthesized result. For the software folks reading this, this is something you'd expect to be able to do in 90 minutes if nothing goes wrong, or maybe if only a few things go wrong.

Subjects were judged purely by how optimized their result was, as long as it worked. Results that didn't pass all tests were disqualified. Although the task was quite simple, it was made substantially more complicated by the strict optimization goal. For any software readers out there, this task is approximately as complicated as implementing the same thing in assembly, where your assembler takes 15-30 minutes to assemble something.

Subjects could use Verilog (unityped) or VHDL (typed). 9 people chose Verilog and 5 chose VHDL.

During the expierment, there were a number of issues that made things easier or harder for some subjects. Overall, Verilog users were affected more negatively than VHDL users. The license server for the Verilog simulator crashed. Also, four of the five VHDL subjects were accidentally given six extra minutes. The author had manuals for the wrong logic family available, and one Verilog user spent 10 minutes reading the wrong manual before giving up and using his intuition. One of the Verilog users noted that they passed the wrong version of their code along to be tested and failed because of that. One of the VHDL users hit a bug in the VHDL simulator.

Of the 9 Verilog users, 8 got something synthesized before the 90 minute deadline; of those, 5 had a design that passed all tests. None of the VHDL users were able to synthesize a circuit in time.

Two of the VHDL users complained about issues with types “I can't believe I got caught on a simple typing error. I used IEEE std_logic_arith, which requires use of unsigned & signed subtypes, instead of std_logic_unsigned.”, and "I ran into a problem with VHDL or VSS (I'm still not sure.) This case statement doesn't analyze: ‘subtype two_bits is unsigned(1 downto 0); case two_bits'(up & down)...' But what worked was: ‘case two_bits'(up, down)...' Finally I solved this problem by assigning the concatenation first to a[n] auxiliary variable."

Comparing mathematical provers; Wiedijk, F

Abstract

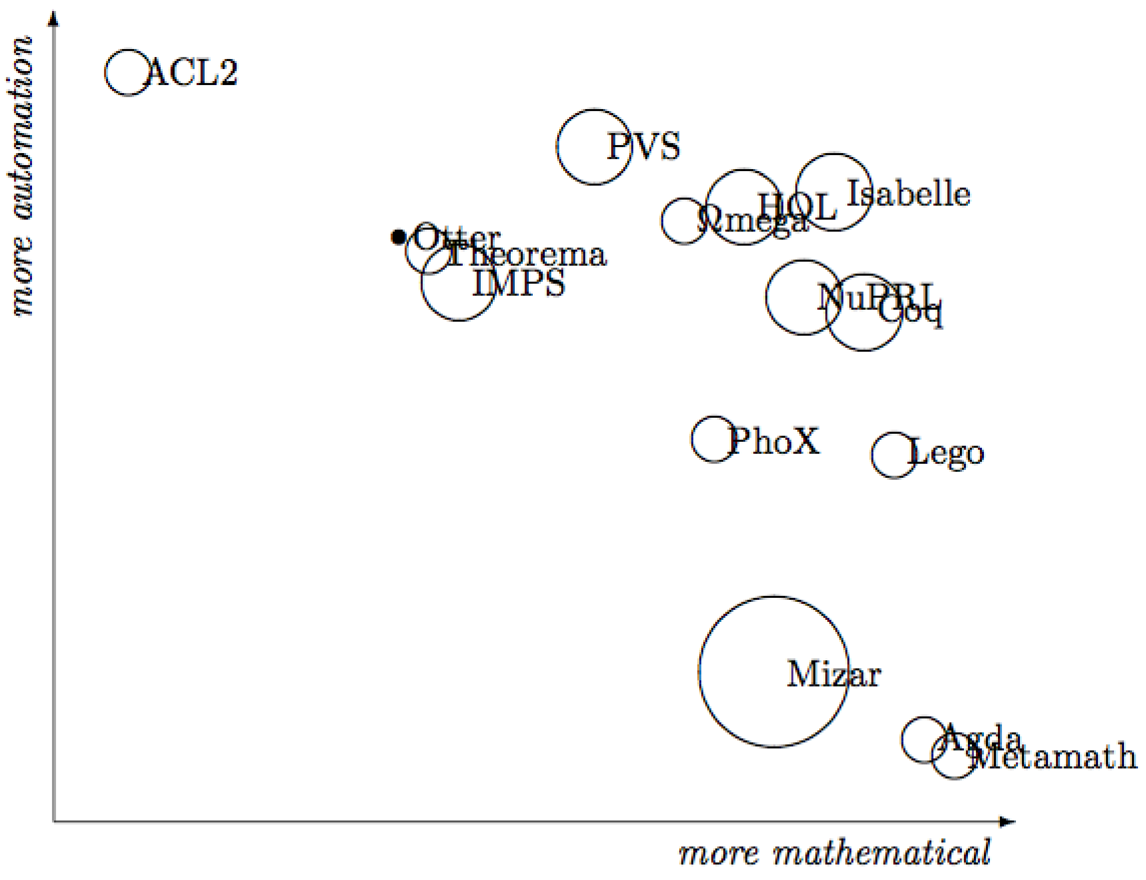
We compare fifteen systems for the formalizations of mathematics with the computer. We present several tables that list various properties of these programs. The three main dimensions on which we compare these systems are: the size of their library, the strength of their logic and their level of automation.

Summary

The author compares the type systems and foundations of various theorem provers, and comments on their relative levels of proof automation.

	HOL	Mizar	PVS	Coq	Otter/Ivy	Isabelle/Isar	Alfa/Agda	ACL2	PhoX	IMPS	Metamath	Theorema	Lego	NuPRL	Ωmega
untyped					•			•			•	•			
decidable non-dependent types	•					•			•	•					•
decidable dependent types		•		•			•						•		
undecidable dependent types			•											•	

	HOL	Mizar	PVS	Coq	Otter/Ivy	Isabelle/Isar	Alfa/Agda	ACL2	PhoX	IMPS	Metamath	Theorema	Lego	NuPRL	Ωmega
primitive recursive arithmetic								•							
first order logic					•										
higher order logic	•		•			•	•		•	•		•	•		•
first order set theory		•				•					•				
higher order type theory				•										•	
classical logic	•	•	•		•	•		•	•	•	•	•			•
constructive logic				•			•						•	•	
quantum logic											•				
fixed logic	•	•	•	•	•		•	•	•	•		•	•	•	•
logical framework						•					•				



The author looked at one particular problem (proving the irrationality of the square root of two) and examined how different systems handle the problem, including the style of the proof and its length. There's a table of lengths, but it doesn't match the updated code examples provided here. For instance, that table claims that the ACL2 proof is 206 lines long, but there's a 21 line ACL2 proof here.

The author has a number of criteria for determining how much automation prover provides, but he freely admits that it's highly subjective. The author doesn't provide the exact rubric used for scoring, but he mentions that a more automated interaction style, user automation, powerful built-in automation, and the Poincare principle (basically whether the system lets you write programs to solve proofs algorithmically) all count towards being more automated, and more powerful logic (e.g., first-order v. higher-order), logical framework dependent types, and de Bruijn criterion (having a small guaranteed kernel) count towards being more mathematical.

Do Programming Languages Affect Productivity? A Case Study Using Data from Open Source Projects; Delory, D; Knutson, C; Chun, S

Abstract

Brooks and others long ago suggested that on average computer programmers write the same number of lines of code in a given amount of time regardless of the programming language used. We examine data collected from the CVS repositories of 9,999 open source projects hosted on SourceForge.net to test this assump- tion for 10 of the most popular programming languages in use in the open source community. We find that for 24 of the 45 pairwise comparisons, the programming language is a significant factor in determining the rate at which source code is written, even after accounting for variations between programmers and projects.

Summary

The authors say “our goal is not to construct a predictive or explanatory model. Rather, we seek only to develop a model that sufficiently accounts for the variation in our data so that we may test the significance of the estimated effect of programming language.” and that's what they do. They get some correlations, but it's hard to conclude much of anything from them.

The Unreasonable Effectiveness of Dynamic Typing for Practical Programs

Abstract

> Some programming language theorists would have us believe that the one true path to working systems lies in powerful and expressive type systems which allow us to encode rich constraints into programs at the time they are created. If these academic computer scientists would get out more, they would soon discover an increasing incidence of software developed in languages such a Python, Ruby and Clojure which use dynamic, albeit strong, type systems. They would probably be surprised to find that much of this software—in spite of their well-founded type-theoretic hubris—actually works, and is indeed reliable out of all proportion to their expectations.This talk—given by an experienced polyglot programmer who once implemented Hindley Milner static type inference for “fun”, but who now builds large and successful systems in Python—explores the disconnect between the dire outcomes predicted by advocates of static typing versus the near absence of type errors in real world systems built with dynamic languages: Does diligent unit testing more than make up for the lack of static typing? Does the nature of the type system have only a low-order effect on reliability compared to the functional or imperative programming paradigm in use? How often is the dynamism of the type system used anyway? How much type information can JITs exploit at runtime? Does the unwarranted success of dynamically typed languages get up the nose of people who write Haskell?

Summary

The speaker used data from Github to determine that approximately 2.7% of Python bugs are type errors. Python's `TypeError`, `AttributeError`, and `NameError` were classified as type errors. The speaker rounded 2.7% down to 2% and claimed that 2% of errors were type related. The speaker mentioned that on a commercial codebase he worked with, 1% of errors were type related, but that could be rounded down from anything less than 2%. The speaker mentioned looking at the equivalent errors in Ruby, Clojure, and other dynamic languages, but didn't present any data on those other languages.

This data might be good but it's impossible to tell because there isn't enough information about the methodology. Something this has going for is that the number is in the right ballpark, compared to the made up number we got when compared the bug rate from Code Complete to the number of bugs found by Farrer. Possibly interesting, but thin.

Summary of summaries

This isn't an exhaustive list. For example, I haven't covered “An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse”, and “Do developers benefit from generic types?: an empirical comparison of generic and raw types in java” because they didn't seem to add much to what we've already seen.

I didn't cover a number of older studies that are in the related work section of almost all the listed studies both because the older studies often cover points that aren't really up for debate anymore and also because the experimental design in a lot of those older papers leaves something to be desired. Feel free to ping me if there's something you think should be added to the list.

Not only is this list not exhaustive, it's not objective and unbiased. If you read the studies, you can get a pretty good handle on how the studies are biased. However, I can't provide enough information for you to decide for yourself how the studies are biased without reproducing most of the text of the papers, so you're left with my interpretation of things, filtered through my own biases. That can't be helped, but I can at least explain my biases so you can discount my summaries appropriately.

I like types. I find ML-like languages really pleasant to program in, and if I were king of the world, we'd all use F# as our default managed language. The situation with unmanaged languages is a bit messier. I certainly prefer C++ to C because std::unique_ptr and friends make C++ feel a lot safer than C. I suspect I might prefer Rust once it's more stable. But while I like languages with expressive type systems, I haven't noticed that they make me more productive or less bug prone[0].

Now that you know what my biases are, let me give you my interpretation of the studies. Of the controlled experiments, only three show an effect large enough to have any practical significance. The Prechelt study comparing C, C++, Java, Perl, Python, Rexx, and Tcl; the Endrikat study comparing Java and Dart; and Cooley's experiment with VHDL and Verilog. Unfortunately, they all have issues that make it hard to draw a really strong conclusion.

In the Prechelt study, the populations were different between dynamic and typed languages, and the conditions for the tasks were also different. There was a follow-up study that illustrated the issue by inviting Lispers to come up with their own solutions to the problem, which involved comparing folks like Darius Bacon to random undergrads. A follow-up to the follow-up literally involves comparing code from Peter Norvig to code from random college students.

In the Endrikat study, they specifically picked a task where they thought static typing would make a difference, and they drew their subjects from a population where everyone had taken classes using the statically typed language. They don't comment on whether or not students had experience in the dynamically typed language, but it seems safe to assume that most or all had less experience in the dynamically typed language.

Cooley's experiment was one of the few that drew people from a non-student population, which is great. But, as with all of the other experiments, the task was a trivial toy task. While it seems damning that none of the VHDL (static language) participants were able to complete the task on time, it is extremely unusual to want to finish a hardware design in 1.5 hours anywhere outside of a school project. You might argue that a large task can be broken down into many smaller tasks, but a plausible counterargument is that there are fixed costs using VHDL that can be amortized across many tasks.

As for the rest of the experiments, the main takeaway I have from them is that, under the specific set of circumstances described in the studies, any effect, if it exists at all, is small.

Moving on to the case studies, the two bug finding case studies make for interesting reading, but they don't really make a case for or against types. One shows that transcribing Python programs to Haskell will find a non-zero number of bugs of unknown severity that might not be found through unit testing that's line-coverage oriented. The pair of Erlang papers shows that you can find some bugs that would be difficult to find through any sort of testing, some of which are severe, using static analysis.

As a user, I find it convenient when my compiler gives me an error before I run separate static analysis tools, but that's minor, perhaps even smaller than the effect size of the controlled studies listed above.

I found the 0install case study (that compared various languages to Python and eventually settled on Ocaml) to be one of the more interesting things I ran across, but it's the kind of subjective thing that everyone will interpret differently, which you can see by looking.

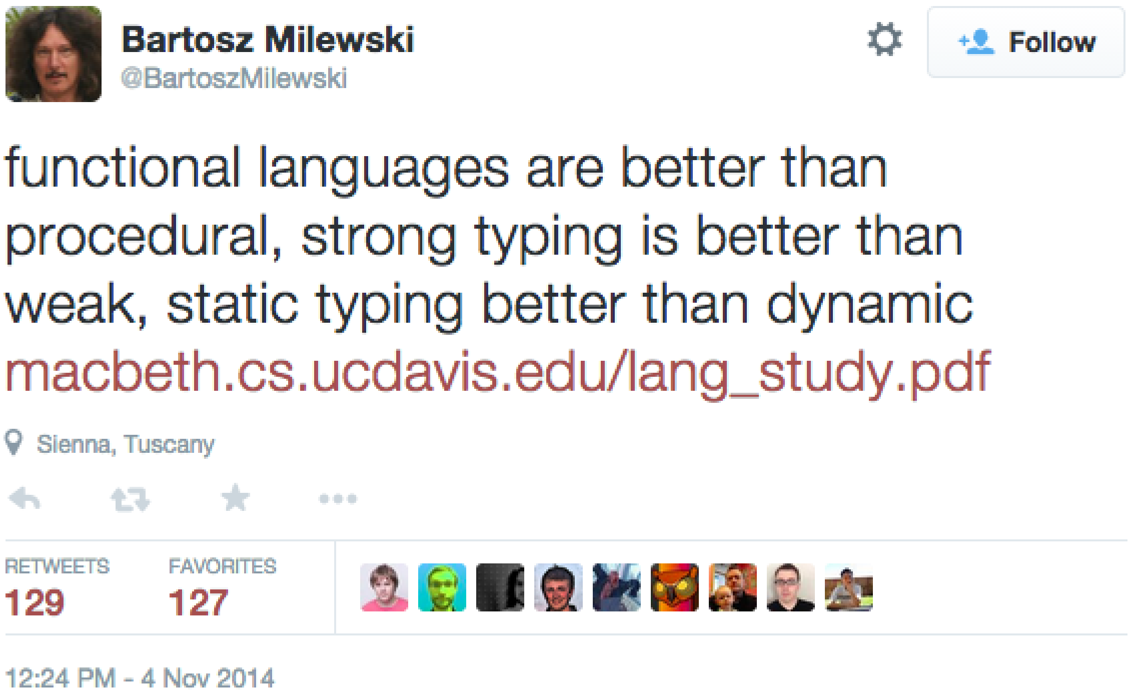
This fits with the impression I have (in my little corner of the world, ACL2, Isabelle/HOL, and PVS are the most commonly used provers, and it makes sense that people would prefer more automation when solving problems in industry), but that's also subjective.

And then there are the studies that mine data from existing projects. Unfortunately, I couldn't find anybody who did anything to determine causation (e.g., find an appropriate instrumental variable), so they just measure correlations. Some of the correlations are unexpected, but there isn't enough information to determine why. The lack of any causal instrument doesn't stop people like Ray et al. from making strong, unsupported, claims.

The only data mining study that presents data that's potentially interesting without further exploration is Smallshire's review of Python bugs, but there isn't enough information on the methodology to figure out what his study really means, and it's not clear why he hinted at looking at data for other languages without presenting the data[2].

Some notable omissions from the studies are comprehensive studies using experienced programmers, let alone studies that have large populations of "good" or "bad" programmers, looking at anything approaching a significant project (in places I've worked, a three month project would be considered small, but that's multiple orders of magnitude larger than any project used in a controlled study), using "modern" statically typed languages, using gradual/optional typing, using modern mainstream IDEs (like VS and Eclipse), using modern radical IDEs (like LightTable), using old school editors (like Emacs and vim), doing maintenance on a non-trivial codebase, doing maintenance with anything resembling a realistic environment, doing maintenance on a codebase you're already familiar with, etc.

If you look at the internet commentary on these studies, most of them are passed around to justify one viewpoint or another. The Prechelt study on dynamic vs. static, along with the follow-ups on Lisp are perennial favorites of dynamic language advocates, and github mining study has recently become trendy among functional programmers.


Bartosz Milewski
@BartoszMilewski

Follow

functional languages are better than procedural, strong typing is better than weak, static typing better than dynamic macbeth.cs.ucdavis.edu/lang_study.pdf

Sienna, Tuscany

RETWEETS 129 FAVORITES 127

12:24 PM - 4 Nov 2014

Other than cherry picking studies to confirm a long-held position, the most common response I've heard to these sorts of studies is that the effect isn't quantifiable by a controlled experiment. However, I've yet to hear a specific reason that doesn't also apply to any other field that empirically measures human behavior. Compared to a lot of those fields, it's easy to run controlled experiments or do empirical studies. It's true that controlled studies only tell you something about a very limited set of circumstances, but the fix to that isn't to dismiss them, but to fund more studies. It's also true that it's tough to determine causation from ex-post empirical studies, but the solution isn't to ignore the data, but to do more sophisticated analysis. For example, econometric methods are often able to make a case for causation with data that's messier than the data we've looked at here.

The next most common response is that their viewpoint is still valid because their specific language or use case isn't covered. Maybe, but if the strongest statement you can make for your position is that there's no empirical evidence against the position, that's not much of a position.

If you've managed to read this entire thing without falling asleep, you might be interested in my opinion on tests.

Responses

Here are the responses I've gotten from people mentioned in this post. Robert Smallshire said "Your review article is very good. Thanks for taking the time to put it together." On my comment about the F# "mistake" vs. trolling, his reply was "Neither. That torque != energy is obviously solved by modeling quantities not dimensions. The point being that this modeling of quantities with types takes effort without necessarily delivering any value." Not having done much with units myself, I don't have an informed opinion on this, but my natural bias is to try to encode the information in types if at all possible.

Bartosz Milewski said "Guilty as charged!". Wow. Much Respect. But notice that, as of this update, The correction has been retweeted 1/25th as often as the original tweet. People want to believe there's evidence their position is superior. People don't want to believe the evidence is murky, or even possibly against them. Misinformation people want to believe spreads faster than information people don't want to believe.

On a related twitter conversation, Andreas Stefik said "That is not true. It depends on which scientific question. Static vs. Dynamic is well studied.", "Profound rebuttal. I had better retract my peer reviewed papers, given this new insight!", "Take a look at the papers...", and "This is a serious misrepresentation of our studies." I muted the guy since it didn't seem to be going anywhere, but it's possible there was a substantive response buried in some later tweet. It's pretty easy to take twitter comments out of context, so check out the thread yourself if you're really curious.

I have a lot of respect for the folks who do these experiments, which is, unfortunately, not mutual. But the really unfortunate thing is that some of the people who do these experiments think that static v. dynamic is something that is, at present, "well studied". There are plenty of equally difficult to study subfields in the social sciences that have multiple orders of magnitude more research going on, that are considered open problems, but at least some researchers already consider this to be well studied!

Acknowledgements

Thanks to Leah Hanson, Joe Wilder, Robert David Grant, Jakub Wilk, Eirenarch, Edward Knight, and Evan Farrer for comments/corrections.

---

1. This was from a talk at Strange Loop this year. The author later clarified his statement with "To me, this follows immediately (a technical term in logic meaning the same thing as “trivially”) from the Curry-Howard Isomorphism we discussed, and from our Types vs. Tests: An Epic Battle? presentation two years ago. If types are theorems (they are), and implementations are proofs (they are), and your SLA is a guarantee of certain behavior of your system (it is), then how can using technology that precludes forbidding undesirable behavior of your system before other people use it (dynamic typing) possibly be anything but unethical?" [return]
2. Just as an aside, I find the online responses to Smallshire's study to be pretty great. There are, of course, the usual responses about how his evidence is wrong and therefore static types are, in fact, beneficial because there's no evidence against them, and you don't need evidence for them because you can arrive at the proper conclusion using pure reason. The really interesting bit is that, at one point, Smallshire presents an example of an F# program that can't catch a certain class of bug via its type system, and the online response is basically that he's an idiot who should have written his program in a different way so that the type system should have caught the bug. I can't tell if Smallshire's bug was an honest mistake or masterful trolling. [return]

← Build uptime *CLWB and PCOMMIT →*

Archive *Support this site (patreon)* *About* *Twitter* *RSS*