Assessing the Type Annotation Burden

John-Paul Ore
Department of Computer Science and Engineering
University of Nebraska–Lincoln, NE, USA
jore@cse.unl.edu

Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska–Lincoln, NE, USA
elbaum@cse.unl.edu

Carrick Detweiler
Department of Computer Science and Engineering
University of Nebraska–Lincoln, NE, USA
carrick@cse.unl.edu

Lambros Karkazis
Department of Computer Science and Engineering
University of Nebraska–Lincoln, NE, USA
lkarkazis@cse.unl.edu

ABSTRACT

Type annotations provide a link between program variables and domain-specific types. When combined with a type system, these annotations can enable early fault detection. For type annotations to be cost-effective in practice, they need to be both accurate and affordable for developers. We lack, however, an understanding of how burdensome type annotation is for developers. Hence, this work explores three fundamental questions: 1) how accurately do developers make type annotations; 2) how long does a single annotation take; and, 3) if a system could automatically suggest a type annotation, how beneficial to accuracy are correct suggestions and how detrimental are incorrect suggestions? We present results of a study of 71 programmers using 20 random code artifacts that contain variables with physical unit types that must be annotated. Subjects choose a correct type annotation only 51% of the time and take an average of 136 seconds to make a single correct annotation. Our qualitative analysis reveals that variable names and reasoning over mathematical operations are the leading clues for type selection. We find that suggesting the correct type boosts accuracy to 73%, while making a poor suggestion decreases accuracy to 28%. We also explore what state-of-the-art automated type annotation systems can and cannot do to help developers with type annotations, and identify implications for tool developers.

CCS CONCEPTS

• **Applied computing** → **Annotation**; • **Software and its engineering** → **Software defect analysis**; • **Theory of computation** → *Type theory*;

KEYWORDS

abstract type inference; physical units; program analysis; static analysis; unit consistency; dimensional analysis; type checking; robotic systems

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238173

ACM Reference Format:
John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. 2018. Assessing the Type Annotation Burden. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238173

1 INTRODUCTION

Type checking is one of the most successful and enduring approaches for ensuring desirable program properties [6, 29, 35, 39, 40]. Indeed, many empirical studies confirm the benefits of type systems [13, 27, 36, 43, 46]. For example, Prechelt *et al.* [36] demonstrated that type checking introduces fewer defects and allows programmers to remove those defects faster, Hannenberg *et al.* [13] claimed static types improve maintainability, and Spiza *et al.* [43] showed that type names alone *even without static type checking* improves the usability of APIs.

Conceptually, type checking consists of three elements: 1) a *type system* to define the abstract theory that can ensure the desired property; 2) a *type mechanism* to enforce type consistency; and, 3) a *type association* to link program variables to their corresponding types. Type association can occur through different means. For common types such as `int`, `float`, or `string` the association is often supported by the programming language and occurs when a variable is declared or is assigned some data of a known type.

For more domain-specific types [12, 18, 31, 53], however, developers must typically incorporate type annotations[1] [6] into the code to link a variable with a type, thereby making the *type association*. Several efforts have explored the benefits of such type annotations. For example, in the context of `JavaScript`, Gao *et al.* [12] found that type annotations help find 15% of bugs in open-source projects. In `Java`, Xiang *et al.* [53] showed the fault detection potential of annotating with *real-world types*, where variables represent measurable quantities in the real world. For `C++`, Ore *et al.* [31] check the physical unit type consistency of files written for the Robot Operating System (ROS) [37] using type associations in a built-in map from class attributes of ROS libraries to physical unit types. For `C`, Jiang and Su [18] checked programs for dimensional unit correctness using lightweight type annotations.

Just like other kinds of code annotation, creating type annotations is a burden for developers, in part because they must first evaluate what program variables need annotation and then choose

[1] 'Type annotations' are sometimes also called 'type hints' [41].

a correct type annotation, hence the name *annotation burden* [7]. But...*how* burdensome? Although many refer to the annotation burden as a given, we lack an understanding of how accurately and quickly developers create type annotations and therefore have difficulty quantifying the benefits to developers.

This work presents an empirical study of 71 subjects to first answer these foundational questions about type annotations:

- RQ_1: How accurately do developers make type annotations?
- RQ_2: How long does a single correct annotation take?

To answer these questions, we design a study where we randomly select code snippets from artifacts in the robotic/cyber-physical domain. We then ask developers to annotate a variable by choosing a physical unit type from a list of common domain types, and to explain why they make the annotation. To our knowledge, this is the first work to quantify the burden in making type annotations, and in general this work contributes to the limited body of data on code annotation.

We instantiate the type annotation task within the domain of physical unit types as identified by Xiang *et al.*'s work on *real-world types* [53]. For example, a variable that represents a *spring constant* in the real-world would be annotated with the physical unit type *newtons-per-meter* ($\mathrm{N\,m^{-1}}$). The type system then checks, for example, that variables of this type are only added or assigned to other variables of this type. We choose this domain because physical unit types are ubiquitous in robotic and cyber-physical software, yet they are nearly always *implicit*, and the lack of explicit typing causes many type inconsistencies [31, 33].

Because of the benefits of annotations, researchers have explored automating the annotation process, including with automated annotation assistants. Vakilian *et al.* [50] found that annotation works best when developers and automated tools work together. We imagine that automatic tools to suggest annotations will continue to improve but occasionally make an incorrect suggestion. Therefore this work also explores the impact of suggesting a type annotation:

- RQ_3: How beneficial to accuracy are correct suggestions and how detrimental are incorrect suggestions?

To address this question, we apply to the study annotation questions one of three treatments: with no suggestion, with a correct suggestion, and with an incorrect suggestion.

The key findings of this work are:

- Developers assign type annotations correctly only 51.4% of the time.
- A developer takes on average 136.0 s to correctly annotate a single variable.
- A correct suggestion reduces the risk of assigning a wrong type by a factor of 0.40, while an incorrect suggestion increases the risk of annotating incorrectly by a factor of 2.66.
- Most subjects cite variable names alone as the clue to explain their annotation, while other subjects reference how reasoning over code operations informs their decisions.
- State-of-the-art tools suggest few correct type annotations, and identifying what variables need to be typed is valuable to developers.

2 BACKGROUND: PHYSICAL UNIT TYPES

This work is instantiated in the type domain of physical units of measure and based on the SI unit system [5]. The SI units system has seven *base units* each of which can itself be a type. Additionally, the seven base units can be combined through multiplication and division to make *compound units*, like *kilogram-meter-squared-per-second-squared* ($\mathrm{kg\,m^2\,s^{-2}}$), more commonly known as *torque*.

Physical units types have been explored in myriad efforts [14, 18, 20, 21, 42, 49, 52]. Not all variables belong to the type domain. For example, `boolean` type variables do not have a physical units type, while `float` variables can and often do represent a measured quantity with real-world meaning, and therefore have a physical unit type in addition to their data type (like `float`). Determining whether a variable belongs in the physical unit type domain is part of the annotation burden.

The type annotation process in the physical unit domain involves extracting clues from variable names, comments, and mathematical constraints. For example, a developer could infer that variable `r` in:

```
dist_meters = r * time_seconds;
```

probably has the type *meters-per-second* ($\mathrm{m\,s^{-1}}$). A competent developer would note that the variables `dist_meters` and `time_seconds` probably have physical units types because of their names, although the name `r` provides little help. Then a developer could solve for the physical unit type of `r` using simple algebra, and perhaps rename `r` to `rate_meters_per_second`. However, sometimes the name does not help, for example, the variable `x` in:

```
x = 0.01; // 1 cm
```

does not hint a type but it likely has the type *meters* (m) because of the comment.

Developers must choose a physical unit type from a large set of possible types, but in practice, some units are more common than others. Table 1 shows the most common unit types ordered by decreasing frequency found in a large corpus of open source robotic code [33].

For our empirical study, we use Table 1 as the list of possible type annotations, with two additions: 1) `NO UNITS`, for scalars and quantities that are not part of the type domain; and 2) `OTHER`, for uncommon types like *kilogram-meter-squared-per-second-cubed-per-ampere* (one of our answers, also known as *voltage*) and to allow subjects to think beyond the choices provided. Including `NO UNITS` is essential because deciding what should be typed is the first part of the annotation burden.

3 METHODOLOGY

In this section, we first describe the type annotation task and reiterate our research questions. We then present our experimental design and discuss how we constructed a test instrument with code artifacts, and how we selected those artifacts. We then describe the target population and how we recruited and pre-screened subjects for the experiment. Next, we explain the experimental phases and finally discuss the tools used during the experiment and analysis.

Table 1: Common physical unit types from [33], in decreasing order of frequency. COVERED denotes whether any question's correct answer on the study was the type listed.

PHYSICAL UNIT TYPE	SI SYMBOL	COVERED
meters	m	✓
second	s	
quaternion	q	✓
radians-per-second	$\mathrm{rad\,s^{-1}}$	✓
meters-per-second	$\mathrm{m\,s^{-1}}$	
radians	rad	✓
meters-per-second-squared	$\mathrm{m\,s^{-2}}$	✓
kilogram-meters-squared-per-second-squared	$\mathrm{kg\,m^2\,s^{-2}}$	✓
meters-squared	$\mathrm{m^2}$	
degrees (360)	deg°	
radians-per-second-squared	$\mathrm{rad\,s^{-2}}$	✓
meters-squared-per-second-squared	$\mathrm{m^2\,s^{-2}}$	✓
kilogram-meter-per-second-squared	$\mathrm{kg\,m\,s^{-2}}$	
kilogram-per-second-squared-per-ampere	$\mathrm{kg\,s^{-2}\,A^{-1}}$	✓
Celsius	°C	
kilogram-per-second-squared	$\mathrm{kg\,s^{-2}}$	✓
kilogram-per-meter-per-second-squared	$\mathrm{kg\,m^{-1}\,s^{-2}}$	
lux	lx	
kilogram-squared-per-meter-squared-per-second-to-the-fourth	$\mathrm{kg^2\,m^{-2}\,s^{-4}}$	

3.1 Type Annotation Task & Research Questions

The type annotation task requires developers to make a type association between a variable and a type. We assess the type annotation task through an online test where we show code snippets to subjects and ask them to choose a physical type annotation for a specified variable. As shown in Figure 1, a test question consists of a code snippet, a highlighted variable, a text question, a suggestion (for some questions), and a drop-down menu of physical unit types. The drop-down box contains 21 type annotations from Table 1 in random order from which subjects must select one. The code snippets used in this study vary in length from 4-57 lines, averaging 17.9 LOC and 2.9 comments as measured by **cloc** [9]. Of the test questions, 14/20 show an entire function while six are truncated so the code snippets fit onto one page. Test questions like the one shown in Figure 1 are instances of the type annotation task that we use to answer three RQs:

- RQ_1: How accurately do developers make type annotations? To answer this question we calculate the percentage of correct responses to a battery of type annotation tasks.
- RQ_2: How long does a single annotation take? To answer this question we measure the time to complete the type annotation task.
- RQ_3: How beneficial are correct suggestions to accuracy and how detrimental are incorrect suggestions? To answer this question we provide a single correct or incorrect suggestion to some questions and measure the change in the percentage of correct responses between questions without suggestions and questions with suggestions.

After the subjects finalize their type annotation through the unit selection, they are asked to provide an open-ended explanation for their choice. We later use the explanations to help us understand how subjects reason about type annotations, both when the type annotation is correct and when incorrect.

3.2 Experimental Design

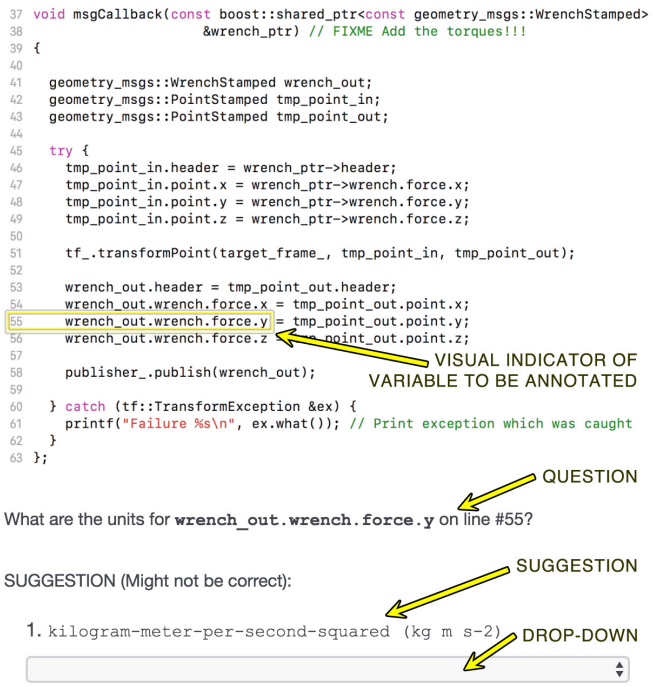


Figure 1: Sample test question. The yellow box on line 55 indicates the variable to be annotated. The test question shows treatment T_2, a correct suggestion.

To address our research questions simultaneously, we design an experiment involving instances of the annotation task described earlier. In our experiment, we measure both response accuracy and the time it takes the subject to select and submit an annotation. Each test question, like the example shown in Figure 1, has one of three treatments:

- T_1: No suggestion (control). A question with the suggestion section not included.
- T_2: Correct suggestion. A question with a correct suggestion immediately above the drop-down box, where the text of the suggestion exactly matches one option in the drop-down. The suggestion is accompanied by the caveat: *"SUGGESTION (Might not be correct)."*
- T_3: Incorrect suggestion. This treatment is identical to T_2 except the suggestion is incorrect. The incorrect suggestion has the same caveat as in T_2 and matches one option in the drop-down box. This incorrect suggestion is chosen randomly from Table 1 (excluding the correct answer and OTHER).

The measurements of accuracy and timing in response to T_1 answer both RQ_1 and RQ_2 and also are the control for RQ_3. To address

RQ_3, we compare the accuracy and time for test questions treated with T_1 to questions treated with T_2 and T_3. In this experiment, the **independent variable** is the suggestion, and the **dependent variables** are annotation accuracy and time.

Our study uses a *completely randomized design* [22]. We group ten questions into an annotation test ('the main test'), and randomly assign subjects to tests. We randomly apply a treatment to each question and randomize question order for each subject. We ensure that each test includes at least three questions of each treatment type, to spread the treatments across subjects.

3.3 Test Instrument Details

Question Timing, Explanations, and Code Artifacts. Questions include a snippet of a code artifact as shown in Figure 1. Each annotation question is instrumented to collect timing information, specifically the duration from when the question is loaded to when the subjects finalize their answer.

As explained earlier, after the subjects finalize their answer to a type annotation test question, we ask them to explain why they chose that type in an open-ended question. We want to record explanations to understand how subjects reason about choosing an annotation type and what differentiates correct from incorrect responses. The time to write the explanation is not included in the time to annotate.

The code artifacts come from a corpus of open-source robotic and cyber-physical code repositories identified in [33] encompassing a wide variety of applications. The corpus contains 797,410 C++ files from 3,484 GitHub repositories that build against the Robot Operating System (ROS) [37], a robotic middleware with many 'real world types.' From those files, we ran the tool Phriky [32] to identify 31,928 files with physical unit type variables. After excluding test files and those that did not compile, we randomly select a file, and starting from the top, we manually identify the first function with unit types and make a judgement about whether the function is sufficiently complex, meaning that it contains either interaction between physical units or compound physical unit types (see Section 2). Within such functions, we randomly select a single variable with a physical unit type. We repeat this process until we have 20 artifacts, and each artifact was reviewed by at least two of the authors. Finally, we cross-check the annotations one more time before the test and one more time during the test instrument evaluation phase. Table 1 shows the resulting distribution of physical unit types within the code artifacts we study.

Suggestions. For treatment T_2, the 21 types in the drop-down menu are the 19 most common physical units found in a corpus of robotic code plus `NO UNITS` and `OTHER` (see Section 2). For treatment T_3, the incorrect suggestion is randomly selected from Table 1 minus the correct answer and `OTHER`. Suggestions are randomized *per test*, so each question has a variety of incorrect suggestions across tests.

Type Annotation Options. At the bottom of Figure 1 is a drop-down menu with annotation type choices. Every question, regardless of treatment, had the same type annotation options in a drop-down menu, with the order randomized for every question to mitigate the threat of response order bias [16].

Tests. We have a pool of 20 artifacts, each with a unique code snippet and correct answer. We compose 20 tests with a different random subset of 10 questions randomly selected from the initial pool of 20. We randomize question order per subject, and randomly assign treatments T_1-T_3 to questions, retaining a balanced number of treatments per test.

A version of the test instrument with all 20 test questions can be found at https://doi.org/10.5281/zenodo.1311901.

3.4 Subject Sample Population

The sample population is users with programming experience recruited using Amazon's Mechanical Turk (MTurk). MTurk is an online marketplace for labor that is increasingly popular for behavior research [25] and has an extensive usage in software engineering [10, 24, 44]. Finding subjects on MTurk is not without risks (especially in demographics) [19] but has been shown to be 'appropriate' for research requiring diverse cognition [17]. The MTurk mechanism allows for 'exclusion criterion' to pre-screen subjects based on a demonstrated ability to complete MTurk tasks successfully.

Following recommended practices [47], we pre-screen subjects by requiring them to have completed >500 tasks with >90% approval in their MTurk history, and further screen subjects by requiring them to pass a pretest of type annotations. We pay subjects a fixed amount for the pretest ($2 USD) and main test ($10 USD) regardless of accuracy, since this has been shown to have little impact on quality among MTurk workers [26]. We tell the subjects not to rush, that they would be judged based on the accuracy of their responses, to provide good explanations, and to watch for random 'attention checks' [15] because this has been shown to increase performance.

Table 2: Demographics for 71 Subjects.

YEARS EXPERIENCE	PROGRAMMING C, C++, C#, Java	EMBEDDED SYSTEMS, CYBER-PHYSICAL, ROBOTICS
< 1	17 (24%)	53 (75%)
1 – 5	37 (52%)	15 (21%)
5+	17 (24%)	3 (4%)

At the beginning of the pretest, we ask three demographic questions about experience with programming languages, robotics, and type annotations. We want to determine if subject demographics would influence responses and to get a sense of who was participating in the study. The first question relates to programming languages: *"How many years of programming experience in languages like* `C, C++, C#, Java`*?"* The second asks about embedded system programming: *"Years of experience programming embedded systems or robotic systems or cyber-physical systems (Things that move or sense)?"* Table 2 shows a summary of the responses for the 71 subjects who completed the main test. As shown in the table, 37/71 (52%) of subjects indicate 1-5 years experience with (mostly) statically typed languages, and 18/71 (25%) have more than one year of experience with robotic or embedded programming. The third question (Yes/No) asks: *"Have you used any code annotation frameworks?"* 13/71 subjects (17%) indicate the previous usage of annotation frameworks and name tools such as `'SAL/MSDN'`, `'Resharper/Jetbrains'`, and `'JSR 308'`. We revisit the impact of these demographic factors in Section 4.

3.5 Tools

In conducting these experiments, we use several off-the-shelf tools:
Amazon's Mechanical Turk (MTurk)[1] is used to recruit and pay subjects both for the pretest and the main tests. We ensure subjects anonymity as required by our Institutional Review Board (IRB# 20170817412EX) by tracking only the MTurk worker ID.
Qualtrics [3] is used to create and deploy the test instruments, randomize test question order per subject, instrument the questions to record timing information, record responses, and generate a unique completion code used to pay subjects. We configure Qualtrics to prevent the same IP address from submitting multiple responses.
R (Statistical Programming Language)[38] is used for data analysis, including the `multinom` function from the `nnet` package [51] to build binomial log-linear response models, and the `binom` package [11] to calculate binomial confidence intervals. We perform ANOVA on timing questions using **R**'s `aov` function.
Clang-format [2] is used to standardize the code formatting of the snippets shown to subjects.

3.6 Study Phases

We conducted the study in two phases:
Phase 1: Test Instrument Evaluation and Refinement. In this phase, we iteratively evaluate the test instruments on 27 subjects, each test with ten questions without suggestions. Based on this evaluation, we: 1) replaced two trivially easy questions; 2) refined the suggestion wording (*"Might not be correct"*) and pretest demographic instructions (*"NOT GRADED OR SCORED,"* as recommended by best practices for MTurk in [19]); 3) verified our annotations; 4) added visual highlights to the variables to be annotated; 5) randomized the question order per test; and, 6) added a required explanation textbox field for every annotation. None of the data acquired in this phase is included in our results, and the 27 evaluation subjects are not eligible to take the main test.

```
16 // Send a velocity command
17 void Stopper::moveForward() {
18   geometry_msgs::Twist
19       msg; // The default constructor will set all commands to 0
20   msg.linear.x = FORWARD_SPEED_MPS;
21   commandPub.publish(msg);
22 }
```

Figure 2: Code snippet used in the pretest.

Phase 2: Test Instrument Deployment of Pretest & Main Test. We require subjects to pass a pretest. In the pretest, all subjects read two practice questions that serve as a tutorial and then must correctly answer two annotation questions. Figure 2 shows a screenshot of a question from the pretest. The correct type assignment, *meters-per-second*, can be inferred from the variable name or the name of the variable assigned to it. In total 1431 subjects started the pretest, but only 487 finished it, indicating that many subjects opted out of the task. Of those that finished the pretest, 30.7% of subjects (145/472) passed the pretest.

After passing the pretest, 49.0% of subjects (71/145) took the main test, which they had to start within 4 hours of the pretest and then had 2 hours to complete. We paid each main-test subject $10 (USD). We received 417 total responses to the main test.[2]

[2]The eagle-eyed reader will notice we have 71 subjects who took a 10 question test, yet have only 417 responses. We had to exclude 293 responses because the test question order was accidentally not randomized early in this phase.

4 RESULTS

We first address the accuracy of responses with no suggestions (T_1, the control treatment) and examine how subjects' demographics impact accuracy, then examine T_1's timing and compare correct responses to incorrect responses. Next, we explore the impact of suggestions on accuracy (treatments T_2 and T_3, respectively) and then examine annotation timing by question difficulty. Finally, we summarize the clues subjects reported using to choose a type.

4.1 RQ_1 Results: Accuracy

For test questions under treatment T_1, with no suggestions, subjects correctly annotate 71/138 (51%), as shown in Figure 3. The figure shows the mean and a 95% binomial proportion confidence interval of ±8.5% (Agresti-Coull) [4].

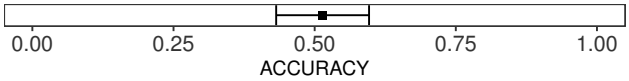


Figure 3: Annotation accuracy for control treatment T_1.

Table 3 shows detailed statistics for accuracy under treatment T_1. As shown in the table, there is a wide range of accuracy based on the question. Based on the accuracy of test questions that received treatment T_1, we grouped questions into three difficulty levels: Easy 100 − 75%, Medium 75 − 25%, Hard 25 − 0%.

Figure 5 shows the range of accuracy for each difficulty group of T_1 (no suggestion). We make this grouping primarily to see if question difficulty correlates to the time necessary to assign a type.
Previous Experience Has Little Impact On Accuracy. Subject's previous experience with programming languages, robotics/cyber-physical systems, and experience with annotations (described in Section 3.4) does not have a significant impact on accuracy. Subjects with 5+ years of programming experience ($N = 17$) have a slightly higher mean accuracy (56%) than other subjects (50% for 1 − 5 years $N = 37$, 50% for < 1 years $N = 17$), but without significance ($p = 0.554$). Surprisingly, subjects with the least experience with robotics/cyber-physical have the highest accuracy (53%, $N = 53$) compared other subjects (45% for 1 − 5 years $N = 15$, 50% for 5+ years $N = 3$), but that is within the noise ($p = 0.829$).
Physical Unit Complexity Has Little Impact On Accuracy. Physical unit types in the SI System have both base unit types (like *meters*) and compound units types (like *kilogram-meters-per-second-squared*), described in Section 2. The increasing complexity of compound types did not appear to correlate to inaccuracy. For example, the correct answer to Q_{19} and Q_{20} is the simple physical unit type *radian*, yet subjects incorrectly annotate this question more than any other, 10/12 times, while the slightly more complex *quaternion* on Q_{12} is never incorrectly annotated, likely because the variable in Q_{12} is assigned from the well-named function `convertPlanarPhi2Quaternion()`. Similarly, compound physical units like *kilogram-meter-squared-per-second-squared* on Q_{17} is incorrect 5/6 times, while the similar compound units *kilogram-per-second-squared* is answered incorrectly 4/10 times in Q_2. Overall, the complexity inherent in the physical unit type seems less important than the surrounding clues, especially good variable names.

RQ_1 Accuracy Results: Manually assigning type annotations is error-prone (51% accurate, ±8.5%).

4.2 RQ_2 Results: Timing

The timing data includes outliers, because our test mechanism has a long time-bound (2 hours). We cap the value of 8/147 timing outliers using Tukey's interquartile 'gate' range method [48]. Tukey's method identifies outliers using a scaling factor k times the interquartile range plus the third quartile, and suggests $k = 3$ [28]. We use an even more conservative $k = 6$ to identify outliers to cap (for upper and lower quartiles $Q1$ and $Q3$, we cap values above $Q3 + k(Q3 - Q1)$ with $k = 6$). In total, we cap question times greater than 961.6 s to the sample mean's 95% value, 529.1 s.

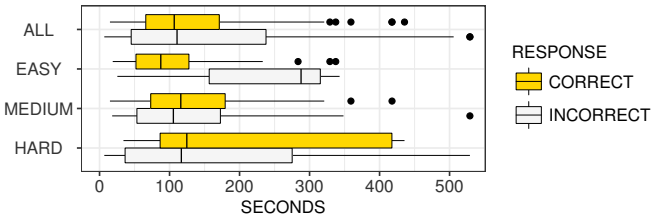


Figure 4: Time to complete a single annotation, separated by question difficulty and correctness annotation.

Subjects take an average of 136.0 s (median=108.6 s) to make a single correct type annotation. The results for annotation times are shown in Figure 4, grouped by question difficulty and correctness. Correct annotations are slightly faster than incorrect ones (mean=169.2 s) but not significantly faster ($p = 0.184$). As shown in the figure, the average time to assign a type annotation (for All) is nearly the same whether correct or incorrect, with slightly more variance for incorrect answers. Hard questions (mean=219.7 s) tend to take longer to answer correctly than Easy questions (mean=112.3 s), but without statistical significance ($p = 0.282$), likely because few Hard questions were answered correctly (we would have needed several more hard questions to have enough statistical power).

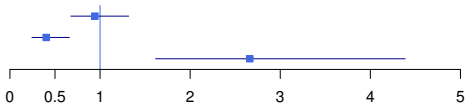
RQ_2 Timing Results: assigning type annotations is time-intensive (mean=136.0 s, median=108.6 s *per variable*).

Table 3: Accuracy and time for questions by treatment.

Q#	DIFFICULTY	CONTROL T_1 NO SUGGESTION Correct %	Correct Fraction	Time (s) Mean	Time (s) Median	TREATMENTS T_2 CORRECT SUGGESTION Correct %	Correct Fraction	Time (s) Mean	Time (s) Median	T_3 INCORRECT SUGGESTION Correct %	Correct Fraction	Time (s) Mean	Time (s) Median
12	Easy	100	6/6	76	70	83	5/6	111	36	33	2/6	162	121
9		90	9/10	113	90	80	8/10	112	70	67	4/6	93	68
5		83	5/6	144	82	83	5/6	237	155	17	1/6	116	49
15		83	5/6	169	141	83	5/6	122	103	40	4/10	125	102
All Easy		89	25/28	124	88	82	23/28	141	70	36	10/28	124	74
4	Medium	67	4/6	153	102	80	8/10	151	105	20	2/10	223	146
6		67	4/6	134	130	75	6/8	156	103	50	3/6	146	76
16		67	4/6	64	65	90	9/10	200	72	33	2/6	104	77
8		64	7/11	130	141	90	9/10	98	79	33	2/6	163	103
2		60	6/10	120	105	33	2/6	75	54	20	2/10	72	58
3		60	6/10	302	233	83	5/6	202	139	17	1/6	150	123
7		50	3/6	226	103	80	8/10	155	153	17	1/6	86	69
10		43	3/7	87	105	83	5/6	97	100	33	2/6	184	184
11		33	2/6	151	128	83	5/6	175	78	67	4/6	107	99
14		33	2/6	106	101	67	4/6	75	42	0	0/6	75	53
18		33	2/6	167	50	100	6/6	126	125	33	2/6	264	218
All Medium		51	41/80	153	112	77	65/84	140	90	28	21/74	143	108
1	Hard	17	1/6	245	188	67	4/6	56	52	40	4/10	258	175
13		17	1/6	130	90	50	3/6	99	67	0	0/6	156	146
17		17	1/6	54	32	33	2/6	198	126	57	4/7	233	111
19		17	1/6	213	201	50	3/6	90	85	17	1/6	174	83
20		17	1/6	234	196	50	3/6	231	168	0	0/6	111	84
All Hard		17	5/30	175	118	50	15/30	135	91	23	8/35	196	99
All Questions		51	71/138	152	109	73	103/142	139	86	28	39/137	153	98

Table 4: Accuracy by treatment. 'Risk Ratio' lines show a multiplication factor indicating the likelihood of an incorrect type annotation with a 95% confidence interval. A Risk Ratio of 2 means twice as likely.

TREATMENT	% CORRECT	RESPONSES	SUBJECTS	RISK RATIO OF INCORRECT TYPE ANNOTATION	*p*-VALUE
T_1 No Suggestion (Control)	51	138	71		-
T_2 Correct Suggestion	73	142	69		0.0003
T_3 Incorrect Suggestion	28	137	58		0.0001



4.3 RQ_3 Results: Impact of Suggestions on Accuracy

RQ_3 considers the impact of a single suggestion on the accuracy and timing of type annotations.

As discussed in Section 3.2, subjects are supplied with a type annotation suggestion immediately below the question text as shown in Figure 1, either correct (T_2) or incorrect (T_3). To measure the significance of the impact of suggestions we fit a binomial log-linear response model ('the model'). We use a binomial response model because the test question responses are either *correct* = 1 or *incorrect* = 0. The output of the model includes the risk ratio by treatment. The risk ratio is used in log-linear models to quantify the likelihood of a binomial response. A risk ratio >1 in our study means an increased risk of assigning an incorrect type when compared to the control (T_1).

As shown in Table 4, a correct suggestion T_2 decreases the risk of annotating incorrectly ($p = 0.0001$) compared to T_1. The model predicts that T_2 reduces the risk of assigning a wrong type by a 0.40 (0.24-0.66, 95% confidence). An incorrect suggestion T_3 increases the risk of annotating incorrectly ($p = 0.0003$) compared to T_1. The model predicts that T_3 increases the risk of assigning a wrong type by 2.66 (1.62-4.39, 95% confidence).

These *p*-values indicate that treatments T_2 and T_3 have a significant impact on annotation accuracy. Treatments T_2 and T_3 are also significantly different from each other ($p = 1.281e-12$).

For treatment T_3 (incorrect suggestion), of the 71 subjects providing 137 responses, 98 responses are incorrect. Of these, 30/98 (31%) responses 'took the bait' of using the provided incorrect suggestion. Regarding subjects, this corresponds to 22/71 (31%) that used an incorrect suggestion for an annotation.

The most common incorrect annotation for T_2 (14/39) and T_3 (28/98) was `NO UNITS`, meaning users infer that the units canceled out or that the correct answer was a scalar. The next most common incorrect annotations are *meters* (T_2 4/39, T_3 12/98) and `OTHER` (T_2 3/39, T_3 7/98).

Figure 5 shows the range of accuracy for all treatments by question difficulty. As shown in the figure, an incorrect suggestion T_3 reduces accuracy for Easy (−53%) and Medium (−49%) questions with little impact on Hard questions. Correct suggestions T_2 benefit all questions compared to T_1, with similar improvements for Hard (+33%) and Medium (+26%) questions, while only helping Easy questions by +7%.

RQ_3 Accuracy Results: Incorrect suggestions increase the risk of incorrect annotations by a factor of 2.66, while correct suggestions reduce the risk of incorrect annotations by a factor of 0.40, an approximately equal but opposite impact on annotation accuracy.

4.4 RQ_3 Results: Impact of Suggestions on Time

Figure 6 shows the impact of a suggestion on the time required to provide a correct annotation. The three difficulty levels are shown along with the category All. For the group All, correct annotations are fastest in T_2 (correct suggestion, mean=126.1 s), compared to 33% longer with T_3 (incorrect suggestions, mean=168.5 s)) and 8% longer with T_1 (no suggestion, mean=136.0 s). The difference between the time between T_2 and T_3 is not significant ($p = 0.220$).

Correct suggestions have the least impact on the timing of Easy questions. This small impact makes sense intuitively since Easy questions benefit less from a suggestion. Correct suggestions tend to

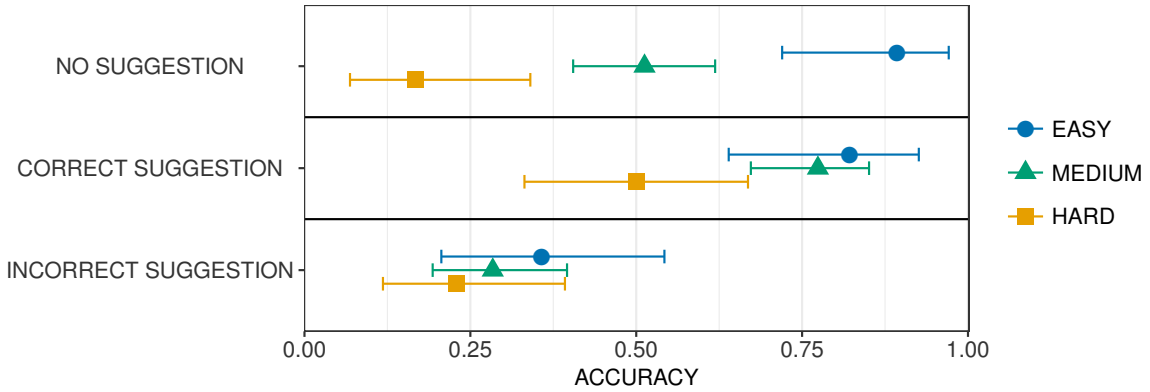


Figure 5: Accuracy by treatment and difficulty, showing 95% confidence interval.

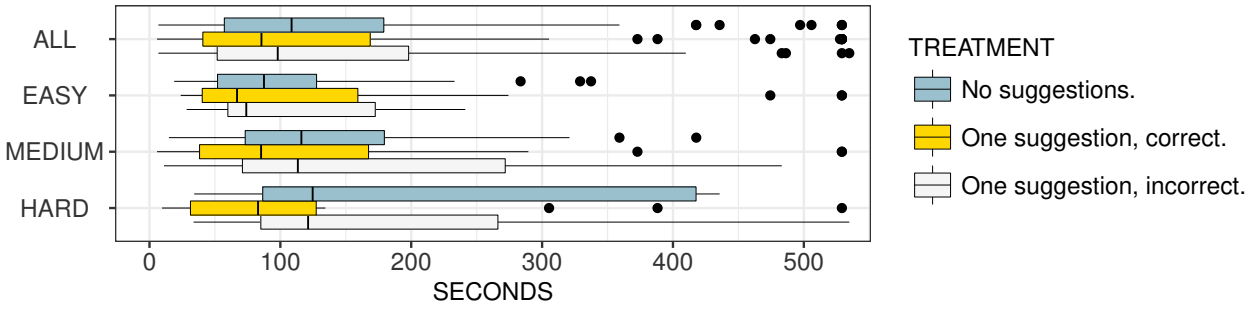


Figure 6: Timing by difficulty and treatment for correct responses.

reduce the time required for HARD questions, as shown in Figure 6, although without statistical significance that we attribute to having few correct HARD answers. Incorrect suggestions T_3 tend to increase the time to annotate both MEDIUM and HARD questions, but without significance.

RQ_3 Timing Results: Although a definitive answer requires further study, correct suggestions appear to decrease the time for correct annotations the most for HARD type annotations.

4.5 Clues for Choosing a Type

Sections 4.1–4.3 provided a quantitative analysis of the responses, revealing an accuracy of 51% which was surprisingly low and which led us to further explore the clues that led developers to choose a particular type. We conducted this exploration by collecting all the explanations provided by the developers on all their responses and analyzing them using a Grounded Theory [45] approach. We categorized the answers based on what we perceived were common patterns, reorganizing the clusters during iterative phases as new and better patterns emerged. During the first iteration, we applied twelve labels. After three iterations, we converged to six clusters and assigned them a label, as shown in Table 5, discriminated by correctness and treatment.

The most common clue used by developers for both, correct and incorrect answers, for T_1 was variable 'names only', used for 71/138 (51%) of all annotations. The caveat is that although all variables had a name, not all of the code snippets included comments or mathematical operations (we discuss them next). But at least from a qualitative point of view, we note that the explanations tended to convey the value of meaningful identifiers:

> *Q_{17}: At least I hope 'torque' is referring to torque.*

Math reasoning and names were frequently used when explaining the correct answers. For example, this is an explanation for a correct answer to the question in Figure 1:

> *Q_4: vx * cos(th) - vy * sin(th) will give a quantity in m / s. Since dt is a quantity in seconds, multiplying by that will yield meters.*

Errors due to poor math reasoning were present but less frequent than we expected. As an example, for the same question we find:

> *Q_4: Meters per second times dt would cause the seconds to cancel out and the meters to square*

Where "cause...the meters to square" is incorrect.

Code comments were also used as effective clues, with more correct ($N = 11$) than incorrect answers ($N = 3$). We note that only two questions (Q_6 and Q_8) contained clues in comments, which may be representative of how common comments are in spite of their value for inferring types. Incorrect answers for treatment T_1 were common for variables not in the type domain (`NO UNITS`) as subjects did not seem able to gather enough clues to determine that the type system even applied.

Five respondents explicitly stated that the suggestions from T_2 helped, and 12 respondents stated that they were misguided by T_3. These values, however, constitute lower bounds as subjects may not have confided us with the full extent to which they use the suggestion. Still, the fact that for T_3 30/98 (31%) of incorrect answers matched the incorrect suggestion we provided illustrates the large potential impact of suggestions on developers' actions.

Qualitative Results: The main clues for type selection are variable names and reasoning over mathematical operations.

Table 5: Summary of type annotation explanations for 417 answers.

EXPLANATION CATEGORY	CORRECT RESPONSES			INCORRECT RESPONSES		
	T_1	T_2	T_3	T_1	T_2	T_3
names only	36	54	17	35	20	44
math reasoning and names	20	24	18	5	4	12
code comments	11	9	2	3	-	-
not in type domain (`NO UNITS`)	4	10	1	19	13	25
used suggestion	-	5	-	-	-	12
type depends on input	-	-	-	5	2	2

5 THREATS TO VALIDITY

5.1 External Threats

Subjects Might Not Represent Developers. We mitigate this hazard by requiring respondents to complete a pretest that at least shows that respondents could understand the task, read code, and correctly identify the physical units that should be assigned to program variables during annotation. Since subjects were not specifically trained to make type annotations, our accuracy measurements likely under-approximates the performance of trained developers.

Annotation Task Fidelity. The annotation task defined in Section 3 with physical unit types in C++ might not generalize other type annotations. First, type systems vary in complexity, and physical unit types might be more or less complicated/time-consuming than all type systems in general. We observe that type annotation requires developer time and involves reasoning about both the type system and how types interact with the code. Second, our study examines type annotations made by non-authors, likely under-approximating our accuracy measurements. Observing code authors could improve fidelity in follow-on work.

Unrepresentative Code Artifacts. The code artifacts might not represent code that needs type annotation more generally. We mitigate this threat by selecting artifacts randomly from a large corpus (5.9 M LOC), although all our samples are for a strongly-typed language (C++). We limited the scope of analysis to a function, the accuracy and time might be different for bigger scopes.

5.2 Internal Threats

Subjects Recruited Through MTurk. We recruited subjects through MTurk, and [19] indicates MTurk subjects might falsify demographic information to participate in online tests. We mitigate this threat by clearly stating on the pretest that demographic question are "NOT GRADED OR SCORED." We also filtered subjects and provided incentives for them to take the task seriously.

Code Context Bias. Bias introduced by our artifacts, including the amount of context or the variety of physical unit types. We mitigate this threat by showing the entire function when feasible and testing the questions during an evaluation phase to make sure it was possible to choose the correct type annotation with the available information.

Test Instrument Format. The question and suggestion format, as provided, does not reflect the full context on how a developer operates in reality and may have affected the subjects in ways we did not anticipate. The test instrumentation and refinement phase helped mitigate this threat.

Physical Unit Type Common Names. Common names for physical unit types like *force* (instead of *kilogram-meters-per-second-squared*) are not an option in the drop-down box. We mitigate this threat by examining every explanation. If subjects identified an equivalent common name for the physical unit type, and answered OTHER or said they could not find the units in the drop-down box, we consider the answer as correct. We considered 7/417 answers as correct because of a common name.

Test Time Window. We measure the time to complete annotation questions but allow a large time window (2 hours) to complete the whole test, during which subjects might take breaks or do other tasks, or take the entire allotted time to find the correct answer ('ceiling effect'), resulting in longer times to answer annotation questions. Therefore our timing values might over-approximate the time to assign a type annotation. We mitigate this threat by identifying and capping timing outliers. More importantly, our timing only captures annotation time, but we recognize that developer's time may be spent in, for example, pursuing leads generated by incorrect annotations.

5.3 Conclusion Threats

Statistical Significance. We do not have enough subjects ($N = 71$) to find statistical significance for some of our hypotheses that have clear trends, because we exhausted the resources available to get more subjects at this time and unanticipated data distribution across some of the factors we considered. For example, the time consumed by questions of different difficulty was not found to be statistically significant because there were few Hard questions with correct responses. Further, this data distribution does not have statistical significance when segmenting responses by demographics. In the future, we will address such limitations by deploying more tests and by monitoring the results to reassign questions to subjects to even out the desired distributions.

6 DISCUSSION AND IMPLICATIONS

Section 4.3 indicated that type annotation suggestions can have a significant impact on accuracy. Building on that insight, we briefly explore the performance of a type annotation tool for the same type domain, physical units. More specifically, we select the open-source tool Phriky [32] (version 1.0.0) that analyzes C++ written for the Robot Operating System (ROS). We selected Phriky because it is open-source, operates on ROS C++ code, is state-of-the-art, and can automatically suggest physical unit types for some program variables. For example, in Figure 2, lines 18-19 define a structure `msg` of type `geometry_msgs::Twist`. Phriky has a lookup table mapping the attributes of this message class to physical units. This mapping enables Phriky to infer in line 20 that the attribute `msg.linear.x` has the physical unit type *meters-per-second*.

Table 6 shows the tool Phriky's physical unit type predictions for each variable that was used in a test question. We obtained these suggestions by running Phriky `--debug-print-ast` on each file containing the code snippets used in the test, and recording what physical unit type is assigned to the variable. As shown in the Table, Phriky makes a suggestion for only 7/20 (35%) of the variables, correct type suggestions for 5/20 (25%) of variables, and incorrect suggestions for 2/20 (10%) of variables. Existing tools have an opportunity to address some of the challenges.

The implications for tools developers include:

- As Figure 5 shows, correct suggestions are most beneficial for Hard type annotations, and therefore tool developers will have a greater impact making correct suggestions in Hard cases.

Table 6: Correct types for each question compared to Phriky units annotations. Ordered by question difficulty.

Q#	Difficulty	Variable Name	Correct Type	Phriky Suggestion
12	Easy	`pose.orientation`	q	✓
9		`delta_d`	m	✓
5		`robotSpeed.angular.z`	$\mathrm{rad\,s^{-1}}$	✗
15		`x2`	$\mathrm{m^2}$	
4	Medium	`delta_x`	m	
6		`w`	$\mathrm{rad\,s^{-1}}$	
16		`av`	$\mathrm{rad\,s^{-1}}$	
8		`path_move_tol_`	m	
2		`springConstant`	$\mathrm{kg\,s^{-2}}$	
3		`ratio_to_consume`	NO UNITS	
7		`x`	NO UNITS	
10		`wrench_out.wrench.force.y`	$\mathrm{kg\,m\,s^{-2}}$	✓
11		`data->gyro_z;`	$\mathrm{m\,s^{-2}}$	✓
14		`xi`	m	
18		`motor_.voltage[1]`	other	
1	Hard	`return`	m	
13		`angular_velocity_covariance`	$\mathrm{rad\,s^{-2}}$	✗
17		`torque`	$\mathrm{kg\,m^2\,s^{-2}}$	✓
19		`anglesmsg.z`	rad	
20		`dyaw`	rad	

- Finding variables that likely need type annotations is valuable because developers struggle to know what variables belong to the type domain.
- Evidence that implies a type might also suggest a new variable name with better type clues.
- Suggest a type only when >50% confident, because incorrect suggestions hurt as much as correct suggestions help.

7 RELATED WORK

Empirical Studies of Types Systems. Several empirical studies confirm the benefits of type systems. Prechelt and Tichy [36] compared the impact of static type checking on student programmers using ANSI C and K&R, where ANSI C's compiler type checked procedure arguments and found fewer defects in programs written with static type checking. Like this work, we are interested in the empirical measurement of types, but unlike this work, we use existing code artifacts instead of new ones. Various efforts [13, 23, 27, 46] claimed static typing improves reliability, maintainability, and understandability of statically typed programs in comparison to dynamic types. While those works weighed the costs and benefits of type systems, we focus on the costs of the type annotation process. Type names, even without type checking, improve the usability of APIs [43], and Rojas and Fraser [8] emphasized the importance of semantically useful names. We likewise find that variable names contain informative clues, but unlike their work, we also find that a misunderstood name can lead to incorrect type annotations.

Type Annotations. Gao, Bird, and Barr [12] examined how type annotations can detect bugs in `JavaScript`, and quantified the annotation burden in terms of a *time tax* and *token tax*. The authors measured their own annotation effort and reported the time and number of tokens to annotate to detect *one bug*. Using their *token tax* (token-annotation-per-bug) and *time tax* (time-per-bug), we infer their time per single annotation to be 127.8 s for TypeScript and 135.8 s for Flow. We measured an uncannily similar 136.0 s for a single type annotation, even though the task, language, skill level, and type domain are different. Like their work, we are interested in the cost of type annotation, but unlike their work, we measure the time for a population of 71 individuals and not the three authors themselves. Xiang *et al.* propose a kind of program analysis with 'Real-World Types' [53]. This analysis requires that an analyst examine all program tokens to decide what needs to be annotated.

Type Annotation Tools and Suggestions. Nimmer and Ernst evaluated the factors that made an annotation assistant useful [30]. Like their work, we perform an empirical evaluation, and unlike their work, we focus on type annotations instead of assertions. The type qualifier tool Cascade shows better results by involving programmers rather than by automatic inference alone [50]. Like their work, we consider automatic inference mixed with developer input to be a natural next step. Parnin and Orso's work on automatic suggestions in fault isolation [34] showed that when a tool makes multiple suggestions to a developer, only the first suggestion is likely to be used. We likewise make only one suggestion and leave for future work a study of multiple suggestions.

8 CONCLUSION

This work contributes a rich characterization of type annotation accuracy and cost. Our findings reveal that user annotations are wrong almost half of the time and that correctly annotating a single variable takes on average more than two minutes. Through a qualitative analysis of the annotation explanations, we find that variable naming and reasoning over the space of operations on the types were the most common culprits of incorrect annotations. Given the challenge of correctly annotating code, there is significant potential for automated tools to reduce this burden; however, they could misguide the developer if the suggestions are incorrect. Further, existing tools that provide such automation only cover a small portion of the annotation space.

In the future, we will broaden the context of this study to include richer kinds of annotations over larger scopes to determine when our findings generalize. This would help to further close the gap in our understanding of the costs and benefits of annotations. We would also like to consider the follow-up phase, when the annotations are consumed by either the developer or another tool, to more precisely understand the cost of incorrect annotations and the number of correct annotations that are needed to receive tangible benefits. Last, we would like to build on existing techniques and tools for automating type suggestions, especially to cover a greater portion of the annotation space and to explore hybrid annotation mechanisms, all while taking into consideration the baseline accuracy and cost identified in this paper.

ACKNOWLEDGMENT

We thank our subjects for taking part in the study. We would also like to thank NIMBUS lab members Urja Acharya, Carl Hildebrandt, Ajay Shankar, and Adam Plowcha for providing feedback on early versions of the type annotation test instrument. This work is supported by NSF award #CCF-1718040.

REFERENCES

[1] 2018. Amazon Mechanical Turk (MTurk). https://www.mturk.com
[2] 2018. Clang: a C language family frontend for LLVM. https://clang.llvm.org
[3] 2018. Qualtrics. https://www.qualtrics.com
[4] Alan Agresti and Brent A Coull. 1998. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician* 52, 2 (1998), 119–126.
[5] BIPM. 2006. *Le Système international d'unités / The International System of Units ('The SI Brochure')* (eighth ed.). Bureau international des poids et mesures. http://www.bipm.org/en/si/si_brochure/
[6] Luca Cardelli. 1996. Type Systems. *ACM Comput. Surv.* 28, 1 (1996), 263–264. https://doi.org/10.1145/234313.234418
[7] Patrice Chalin and Perry R. James. 2007. Non-null References by Default in Java: Alleviating the Nullity Annotation Burden. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. 227–247. https://doi.org/10.1007/978-3-540-73589-2_12
[8] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating unit tests with descriptive names or: would you name your children thing1 and thing2?. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 57–67. https://doi.org/10.1145/3092703.3092727
[9] Al Danial. 2018. Count Lines Of Code. https://github.com/AlDanial/cloc
[10] Rafael Maiani de Mello, Pedro Correa da Silva, Per Runeson, and Guilherme Horta Travassos. 2014. Towards a Framework to Support Large Scale Sampling in Software Engineering Surveys. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, New York, NY, USA, Article 48, 4 pages. https://doi.org/10.1145/2652524.2652567
[11] Sundar Dorai-Raj. 2014. *binom: Binomial Confidence Intervals For Several Parameterizations.* https://CRAN.R-project.org/package=binom R package version 1.1-1.
[12] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 758–769. https://doi.org/10.1109/ICSE.2017.75
[13] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* 19, 5 (2014), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1
[14] Sudheendra Hangal and Monica S. Lam. 2009. Automatic dimension inference and checking for object-oriented programs. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 155–165. https://doi.org/10.1109/ICSE.2009.5070517
[15] David J Hauser and Norbert Schwarz. 2016. Attentive Turkers: MTurk participants perform better on online attention checks than do subject pool participants. *Behavior research methods* 48, 1 (2016), 400–407.
[16] Allyson Holbrook. 2011. *Encyclopedia of Survey Research Methods.* Sage Publications, Inc., Chapter Response Order Effects. https://doi.org/10.4135/9781412963947
[17] Ronnie Jia, Zachary R. Steelman, and Blaize Horner Reich. 2017. Using Mechanical Turk Data in IS Research: Risks, Rewards, and Recommendations. *CAIS* 41 (2017), 14. http://aisel.aisnet.org/cais/vol41/iss1/14
[18] Lingxiao Jiang and Zhendong Su. 2006. Osprey: a practical type system for validating dimensional unit correctness of C programs. In *28th International Conference on Software Engineering, ICSE 2006), Shanghai, China, May 20-28, 2006*. 262–271. https://doi.org/10.1145/1134323
[19] Irene P. Kan and Anna Drummey. 2018. Do imposters threaten data quality? An examination of worker misrepresentation and downstream consequences in Amazon's Mechanical Turk workforce. *Computers in Human Behavior* 83 (2018), 243–253. https://doi.org/10.1016/j.chb.2018.02.005
[20] Michael Karr and David B. Loveman, III. 1978. Incorporation of Units into Programming Languages. *Commun. ACM* 21, 5 (May 1978), 385–391. https://doi.org/10.1145/359488.359501
[21] Andrew Kennedy. 2009. Types for Units-of-Measure: Theory and Practice. In *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*. 268–305. https://doi.org/10.1007/978-3-642-17685-2_8
[22] Roger E Kirk. 1982. *Experimental design.* Wiley Online Library.
[23] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. Do static type systems improve the maintainability of software systems? An empirical study. In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*. 153–162. https://doi.org/10.1109/ICPC.2012.6240483
[24] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. 2017. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software* 126 (2017), 57 – 84. https://doi.org/10.1016/j.jss.2016.09.015
[25] Winter Mason and Siddharth Suri. 2012. Conducting behavioral research on AmazonâĂŹs Mechanical Turk. *Behavior research methods* 44, 1 (2012), 1–23.
[26] Winter A. Mason and Duncan J. Watts. 2009. Financial incentives and the "performance of crowds". *SIGKDD Explorations* 11, 2 (2009), 100–108. https://doi.org/10.1145/1809400.1809422
[27] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, Tucson, AZ, USA, October 21-25, 2012*. 683–702. https://doi.org/10.1145/2384616.2384666
[28] Robert McGill, John W Tukey, and Wayne A Larsen. 1978. Variations of box plots. *The American Statistician* 32, 1 (1978), 12–16.
[29] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4
[30] Jeremy W. Nimmer and Michael D. Ernst. 2002. Invariant Inference for Static Checking: An Empirical Evaluation. *SIGSOFT Softw. Eng. Notes* 27, 6 (Nov. 2002), 11–20. https://doi.org/10.1145/605466.605469
[31] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. Lightweight Detection of Physical Unit Inconsistencies Without Program Annotations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 341–351. https://doi.org/10.1145/3092703.3092722
[32] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. Phriky-units: A Lightweight, Annotation-free Physical Unit Inconsistency Detection Tool. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 352–355. https://doi.org/10.1145/3092703.3098219
[33] J. P. Ore, S. Elbaum, and C. Detweiler. 2017. Dimensional inconsistencies in code and ROS messages: A study of 5.9M lines of code. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 712–718. https://doi.org/10.1109/IROS.2017.8202229
[34] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. 199–209. https://doi.org/10.1145/2001420.2001445
[35] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.
[36] Lutz Prechelt and Walter F. Tichy. 1998. A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Trans. Software Eng.* 24, 4 (1998), 302–312. https://doi.org/10.1109/32.677186
[37] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3.2. Kobe, Japan, 5.
[38] R Core Team. 2013. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/
[39] Baishakhi Ray, Daryl Posnett, Premkumar T. Devanbu, and Vladimir Filkov. 2017. A large-scale study of programming languages and code quality in GitHub. *Commun. ACM* 60, 10 (2017), 91–100. https://doi.org/10.1145/3126905
[40] John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science)*, Bernard Robinet (Ed.), Vol. 19. Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148
[41] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP484 – Type Hints. https://www.python.org/dev/peps/pep-0484/. [Online; accessed 13-July-2018].
[42] G. Rosu and Feng Chen. 2003. Certifying measurement unit safety policy. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* 304–309. https://doi.org/10.1109/ASE.2003.1240326
[43] Samuel Spiza and Stefan Hanenberg. 2014. Type names without static type checking already improve the usability of APIs (as long as the type names are correct): an empirical study. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*. 99–108. https://doi.org/10.1145/2577080.2577098
[44] Kathryn T. Stolee and Sebastian Elbaum. 2010. Exploring the Use of Crowdsourcing to Support Empirical Studies in Software Engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*. ACM, New York, NY, USA, Article 35, 4 pages. https://doi.org/10.1145/1852786.1852832
[45] Anselm Strauss and Juliet M Corbin. 1990. *Basics of qualitative research: Grounded theory procedures and techniques.* Sage Publications, Inc.
[46] Andreas Stuchlik and Stefan Hanenberg. 2011. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages, DLS 2011, October 24, 2011, Portland, OR, USA*, Theo D'Hondt (Ed.). ACM, 97–106. https://doi.org/10.1145/2047849.2047861

[47] Kyle A. Thomas and Scott Clifford. 2017. Validity and Mechanical Turk: An assessment of exclusion methods and interactive experiments. *Computers in Human Behavior* 77 (2017), 184–197. https://doi.org/10.1016/j.chb.2017.08.038
[48] John W Tukey. 1977. *Exploratory data analysis*. Vol. 2. Reading, Mass.
[49] Zerksis D. Umrigar. 1994. Fully static dimensional analysis with C++. *SIGPLAN Notices* 29, 9 (1994), 135–139. https://doi.org/10.1145/185009.185036
[50] Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. 2015. Cascade: A Universal Programmer-Assisted Type Qualifier Inference Tool. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 234–245. https://doi.org/10.1109/ICSE.2015.44
[51] W. N. Venables and B. D. Ripley. 2002. *Modern Applied Statistics with S* (fourth ed.). Springer, New York. http://www.stats.ox.ac.uk/pub/MASS4 ISBN 0-387-95457-0.
[52] Mitchell Wand and Patrick O'Keefe. 1991. Automatic Dimensional Inference. In *Computational Logic - Essays in Honor of Alan Robinson*, Jean-Louis Lassez and Gordon D. Plotkin (Eds.). The MIT Press, 479–483.
[53] Jian Xiang, John Knight, and Kevin Sullivan. 2015. Real-World Types and Their Application. In *Proceedings of the 34th International Conference on Computer Safety, Reliability, and Security - Volume 9337 (SAFECOMP 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 471–484.