An Empirical Study on Type Annotations: Accuracy, Speed, and Suggestion Effectiveness

JOHN-PAUL ORE, North Carolina State University
CARRICK DETWEILER, University of Nebraska–Lincoln
SEBASTIAN ELBAUM, University of Virginia

Type annotations connect variables to domain-specific types. They enable the power of type checking and can detect faults early. In practice, type annotations have a reputation of being burdensome to developers. We lack, however, an empirical understanding of how and why they are burdensome. Hence, we seek to measure the baseline accuracy and speed for developers making type annotations to previously unseen code. We also study the impact of one or more type suggestions. We conduct an empirical study of 97 developers using 20 randomly selected code artifacts from the robotics domain containing physical unit types. We find that subjects select the correct physical type with just 51% accuracy, and a single correct annotation takes about 2 minutes on average. Showing subjects a single suggestion has a strong and significant impact on accuracy both when correct and incorrect, while showing three suggestions retains the significant benefits without the negative effects. We also find that suggestions do not come with a time penalty. We require subjects to explain their annotation choices, and we qualitatively analyze their explanations. We find that identifier names and reasoning about code operations are the primary clues for selecting a type. We also examine two state-of-the-art automated type annotation systems and find opportunities for their improvement.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Data types and structures**; **Abstract data types**; *Software reliability*; • **Theory of computation** → *Program analysis*;

Additional Key Words and Phrases: Type checking, automated static analysis, software reliability, annotations, program analysis, dimensional analysis, physical units, robotic systems

ACM Reference format:
John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2021. An Empirical Study on Type Annotations: Accuracy, Speed, and Suggestion Effectiveness. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 20 (February 2021), 29 pages.
https://doi.org/10.1145/3439775

1 INTRODUCTION

Type checking is one of the best and time-tested methods of determining if a program has desirable properties [8, 57, 62, 63]. As Milner said, "well-typed programs cannot go wrong" [48]. Many

This work is supported in part by NSF awards IIS-1638099, CCF-1718040, CCF-1853374, and CCF-1909414, and by USDA-NIFA 2017-67021-25924.
Authors' addresses: J.-P. Ore, North Carolina State University, 890 Oval Drive, Raleigh, NC, 27695; email: jwore@ncsu.edu; C. Detweiler, University of Nebraska–Lincoln, 256 Avery Hall, Lincoln, NE, 68588; email: carrick@cse.unl.edu; S. Elbaum, University of Virginia, Rice 423, PO Box 400740, Charlottesville, VA, 22904; email: selbaum@viginia.edu.

1049-331X/2021/02-ART20 $15.00
https://doi.org/10.1145/3439775

studies have empirically measured type checking's benefits for software quality. For example, Prechelt and Tichy [58] showed that developers using type checking wrote programs with fewer bugs and fixed those bugs faster. Spiza and Hanenberg [69] demonstrated that using type names alone helps an API's usability, even with no type enforcement mechanism. Hanenberg et al. [24] showed that programs using static type checking are easier to maintain.

At a high level, type checking has three parts:

(1) A *type system* specifies how objects of different types may interact.
(2) An *enforcement mechanism* ensures the typed program conforms to the type system's rules.
(3) A *type association* connects identifiers in the program to a type in the type system.

The third part of type checking—connecting identifiers to types—happens either during code creation or evolution. During code creation, a variable can be associated with a type during declaration or assignment using types supported by many typed languages such as `string`, `float`, and `int`. During code evolution, additional type associations can be added that extend type safety. Adding type associations during evolution often requires type annotations.

Type annotations can bring type checking to domain-specific or legacy code. For example, Facebook code maintainers (who might not be the code's authors) gradually increased the type safety of a legacy PHP codebase by applying type annotations [8] with the `Hack` programming language [17]. The goal of increased type safety also motivates `Typescript` [61] and gradually typed `Python` [79]. For domain-specific code, many efforts explored the effectiveness of domain-specific type checking through type annotations [9, 19, 31, 50, 81].

The problem is that, in practice, type annotations have a reputation of being burdensome, like code annotations more generally. It is burdensome for several reasons, including that developers must first determine what needs to be annotated and then associate a correct type, all part of the *annotation burden* [9]. But… *how* burdensome? In spite of the "common knowledge" that making annotations is burdensome, we lack an empirical baseline measurement of accuracy and timing, the impact of type suggestions, and the reasons developers give for making type annotations. By characterizing the factors that influence the difficulty of making annotations, this work aims to help researchers and tool builders better target future solutions.

This work presents an empirical study of 97 subjects to answer these questions about type annotations in previously unseen code:

RQ$_1$ How accurately do subjects associate types?
RQ$_2$ How quickly can subjects make correct type annotations?

To address these questions, we design an empirical study where we show subjects a code snippet with a variable that might require a type annotation. In our study, subjects choose a type annotation for the indicated variable from a drop-down list of frequently occurring domain types, and then subjects are required to explain why they chose a type.

We consider type annotations within the domain of physical unit types [81], such as *meter-per-second-squared* ($\mathrm{m\,s^{-2}}$). For example, a variable used to store readings from an IMU sensor might be annotated with the physical unit type $\mathrm{m\,s^{-2}}$. This allows a type enforcement mechanism to guarantee, for example, that variables of $\mathrm{m\,s^{-2}}$ type are only added to variables of $\mathrm{m\,s^{-2}}$ type. We selected the type domain of physical units because cyber-physical and robot code, including the code we have developed and maintained in our robot systems, contains variables that have a physical unit type, but these types are often *latent* or *implicit*, allowing type inconsistencies to go undetected [31, 50, 52, 82]. Also, we found that it is not difficult to recruit subjects who are familiar with physical units. We note that developers must still reason about types and type interactions **no matter the type domain**.

In practice, developers adding type annotations will likely make multiple annotations in the same sections of a program and grow more familiar with it. Since the previous questions ask subjects to make type annotations to previously unseen code, we also want to explore if subjects who make several type annotations to the same code snippet become more accurate or faster, so we ask:

RQ$_3$ Do subjects quickly improve in accuracy or speed when making several related type annotations in the same program?

Since type annotations enable type checking's proven benefits, researchers have explored various ways to automate the type annotation process. Several efforts [18, 77] showed that developers and automated tools working together outperformed automated tools alone. Tools can quickly narrow the hypothesis space but often provide suggestions that are uncertain and fallible. Similarly, since a tool's type suggestions cannot be certain, we would like to understand how such suggestions affect a developer's annotation decision. We find no previous work measuring the impact of suggestions, either correct or incorrect. Therefore, we ask:

RQ$_4$ What is the impact of correct and incorrect suggestions? How does the impact of a single suggestion compare to three suggestions?

To better understand *how* and *why* developers associate a type, we pose a qualitative question:

RQ$_5$ Why do developers choose a particular type?

We address this question by requiring subjects to provide a detailed explanation of each type annotation and organize their responses using Open Coding [47].

Our findings are:

- Subjects' type annotation accuracy is only 51%.
- It takes more than 2 minutes to make the first correct type annotation to previously unseen code (135 s) and 79 s for each subsequent correct annotation.
- Suggestions have a strong impact on annotation accuracy, with a single suggestion increasing accuracy to 73% when correct and reducing accuracy to 28% when incorrect.
- Providing three suggestions is better than one suggestion because three incorrect suggestions are less harmful than a single incorrect suggestion, and three suggestions (correct first) benefit accuracy nearly as much as one correct suggestion.
- Multiple suggestions, though significantly beneficial when correct, do not significantly increase the time to make a correct annotation.
- Identifier names are the main reason for associating a type, both for correct and incorrect annotations, while reasoning about the abstract domain together with identifier names is more likely to be credited for correct annotations.
- Developers need help identifying the variables that need to be typed.
- State-of-the-art tools suggest few correct type annotations.

This article extends our conference paper [53] in several ways. First, we conduct two follow-on studies: (1) a study measuring the impact of multiple suggestions, yielding new results on accuracy and explanations, and (2) a study assessing the impact of making several type annotations within the same code snippet to determine if the accuracy or timing changes as subjects become more familiar with the code. Second, we analyze artifacts' code attributes, such as whether they use high-quality identifiers, and provide new empirical evidence of the benefits of high-quality

Table 1. The Most Common Physical Unit Types Found in a Large Corpus of Open-Source Robot Software [52], in Decreasing Order of Frequency

PHYSICAL UNIT TYPE	DESCRIPTION	SYMBOL	COVERED
meters	distance	m	✓
second	time	s	
quaternion	3-D rotation	q	✓
radians-per-second	angular velocity	$\mathrm{rad\ s^{-1}}$	✓
meters-per-second	linear velocity	$\mathrm{m\ s^{-1}}$	
radians	2-D rotation	rad	✓
meters-per-second-squared	acceleration	$\mathrm{m\ s^{-2}}$	✓
kilogram-meters-squared-per-second-squared	torque	$\mathrm{kg\ m^2\ s^{-2}}$	✓
meters-squared	area	$\mathrm{m^2}$	✓
degrees (360°)	rotation	deg°	
radians-per-second-squared	angular acceleration	$\mathrm{rad\ s^{-2}}$	✓
meters-squared-per-second-squared	velocity covariance	$\mathrm{m^2\ s^{-2}}$	✓
kilogram-meter-per-second-squared	force	$\mathrm{kg\ m\ s^{-2}}$	
kilogram-per-second-squared-per-ampere	magnetic flux density	$\mathrm{kg\ s^{-2}\ A^{-1}}$	✓
Celsius	temperature	$^\circ\mathrm{C}$	
kilogram-per-second-squared	spring constant	$\mathrm{kg\ s^{-2}}$	✓
kilogram-per-meter-per-second-squared	air pressure	$\mathrm{kg\ m^{-1}\ s^{-2}}$	
lux	luminous emittance	lx	
kilogram-squared-per-meter-squared-per-second-to-the-fourth	force covariance	$\mathrm{kg^2\ m^{-2}\ s^{-4}}$	

COVERED denotes whether our study used that physical unit type as a correct answer.

identifiers to annotation type accuracy. Third, we expand our analysis on type suggestion tools with a comparison of two state-of-the-art tools: PHRIKY [51] and PHYS [34]. Lastly, we increase the detail throughout, including the study design, implementation, and an extended related work section.

2 BACKGROUND

In this section, we provide background about the SI System, physical unit types, and how these types are used in programs. We define the type annotation task and the type annotation burden.

2.1 SI System and Physical Units

The physical unit types considered in this work are based on the SI unit system [6]. This system includes the familiar *kilogram*, *meter*, and *second* that represent quantities with dimensions *mass*, *distance*, and *time*, respectively. These units, together with four more: *mols*, *amperes*, *Kelvin*, and *candela*, are the seven *base units*. Every physical phenomenon[1] that can be measured can be quantified in terms of some combination of the seven base units of the SI System. Units in the SI system have a constructive nature: multiplying or dividing the base units constructs new units, called *compound units*, such as *meter-per-second* or *kilogram-meter-squared-per-second-squared* (known colloquially as velocity and Newton-meters, respectively). All units, both base and compound, can be unit types. In practice, some unit types are more common than others. Table 1 shows the most common physical unit types found in a large corpus (5.9 *MLOC*) of open-source robotics code [52], to be discussed in more detail in Section 3.4.

[1]Some subatomic phenomena, like quantum chromodynamics, are not expressed in the SI system because they can be inferred but not directly measured.

2.2 Physical Units in Programs as Types

In some programs, quantities represent real-world phenomena. These quantities are stored in variables that can be explicitly associated with a unit type during declaration or assignment, providing that the language, extension, or library supports physical units. Otherwise, physical quantities in programs can *implicitly* represent real-world phenomena and be represented in the program with more general data types like `float` or `double` (or less commonly, `int`). This implicit representation is the kind of scenario explored in this work. If a developer wants to ensure consistent manipulation of these quantities, he or she must add an explicit *type annotation* in the program so that a type enforcement mechanism knows what to type check. Determining what type annotation, if any, should be made for each program variable requires developer effort to perform the task of type annotation.

2.3 Type Annotation Task and Type Annotation Burden

When a developer seeks to associate a program identifier with a type, he or she can do so in several ways, such as type support libraries, languages, or language extensions. For all these ways, developers must add extra information to associate an identifier to the correct type. Ideally, a developer might have *a priori* knowledge of every identifier's type, but this is not always the case. In many situations, developers determine the correct type by reasoning about interactions between types and using the available evidence to infer a type for a variable.

We define the **type annotation task** as follows. Let T be the set of types in some type domain and V be the set of program variables. Then the type annotation task is to find the function f that maps from program variables to types, such that

$$f : \mathsf{V} \rightarrow \mathsf{T} \tag{1}$$

We assume the set of types T contains the empty element ϵ to account for the case when a program variable does not have a type. Finding the type annotation function f is usually a manual process and requires developers to find evidence to link program variables to types.

There are at least four kinds of code evidence developers could use to reason about types: variable names, comments, code operations, and context. For example, in the code

```
linVel = 0.42;
```

the variable name `linVel` provides a hint that this variable represents a linear velocity with physical unit type *meter-per-second* (ms^{-1}) because it contains the substrings `lin` and `Vel`.

Code comments can also contain clues. Consider the comment following this code:

```
goal_tolerance = 0.01; // one cm
```

The comment `one cm`, which might stand for *centimeter*, together with the value `0.01` provides evidence that `goal_tolerance`'s type is *meter* (m).

Code operations show how variables interact with respect to the type domain. In the code

```
x = x_vel * duration;
```

the physical unit type of `x` is inferred from evidence in the expression on the right-hand side as the result of the multiplication expression. If `x_vel`'s type is linear velocity measured in *meter-per-second* (ms^{-1}) and `duration`'s type is *second* (s), then `x's` type must be *meter* (m).

The context surrounding a variable can provide useful clues for types. For example, domain-specific libraries can define data structures with domain-defined physical unit types that, when used with other code, create a context in which other variables' types can be inferred. This kind of contextual clue is used by the type inference tool Phriky [51]. Variables that interact with shared libraries' data structures can then be inferred by flow. These contexts are limited in that not all program variables come from or interact with shared libraries.

Contexts, code operations, comments, and identifier names help developers determine a unit type for a variable, but not all variables have a corresponding type in the type domain (their type is ϵ, the empty element). For example, Boolean values (`bool`) and program counters (`int`) do not have a physical unit type. Some values are *scalars*, i.e., magnitudes without physical units.

Determining whether an identifier has a unit type is the first part of the annotation burden, followed by associating the correct unit type. We denote the developer's effort of time and energy to perform the type annotation task as the *type annotation burden*.

Definition 2.1. **Type Annotation Burden:** The time and effort spent by developers to associate identifiers to types.

3 METHODOLOGY

In this section, we describe both our research questions and how we address these questions with an experiment using a test instrument made from code artifacts. We discuss the experimental design by first showing how we find code artifacts from open-source repositories, then how code artifacts become test questions, and how subjects are recruited. Finally, we describe the phases of the study.

3.1 Research Questions

To better understand how developers make type annotations, we pose several research questions. By answering these questions, we seek empirical evidence for the accuracy and timing of the type annotation burden, the impact of suggestions, and the reasons developers make type annotations.

RQ$_1$ How accurately do subjects associate types? This question seeks to measure developers' accuracy associating types to program identifiers in the context of previously unseen code. We do this for two reasons: first, to determine if there is empirical evidence supporting the claim that the type annotation task is difficult for developers, and second, to establish a baseline accuracy for developers to make type annotations without automated support, which helps quantify the expected utility of a tool that automatically suggests types.

RQ$_2$ How quickly can subjects make correct type annotations? This question helps us better assess how much time is required to correctly associate a type to an identifier. From this measurement, we might extrapolate the time required to annotate whole programs and better understand the type annotation burden's temporal dimension.

RQ$_3$ Do subjects quickly improve in accuracy or speed when making several related type annotations in the same program? This question helps us better understand how accuracy and timing change during a process of making multiple annotations, as developers would likely do in practice. To answer it, we examined subjects' accuracy and speed when making multiple related annotations in one code region. The annotations are related because they are for either variables involved in the same right-hand side expressions or a transitive assignment. The code artifacts used in this experiment are six artifacts selected from the 20 used in the main test. The results are based on 14 subjects' responses. We show each subject three code snippets. We chose these artifacts because they had multiple variables involved in expressions.

RQ$_4$ What is the impact of correct or incorrect suggestions? How does the impact of a single suggestion compare to three suggestions? Suggestions are important because we believe that developers will increasingly work together with type annotation tools to infer and suggest types. We imagine these kinds of tools might use sources of evidence with various degrees of

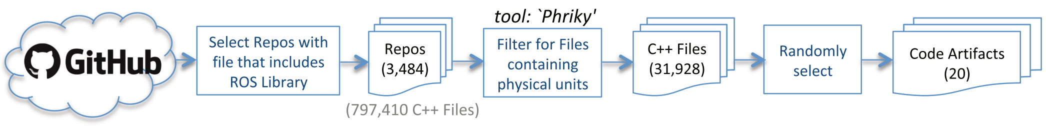


Fig. 1. Process by which code artifacts are selected from the code corpus.

certainty, such as domain knowledge, identifiers, comments, and context [20]. These sources could provide useful clues but are uncertain. Additionally, we ask about the impact of three suggestions, to determine if there is a difference between the effects of a single suggestion and multiple suggestions. We choose three because previous work suggests developers consider only the top few recommendations [55].

RQ$_5$ Why do developers choose a particular type? Unlike the previous research questions, this question is qualitative and asks *why* and *how* developers choose a type. In Section 2.3, we identify possible sources of information developers might use to determine a type, but this question seeks to elicit the developer's reasoning and thought process for making particular type associations. Once subjects have selected a type annotation, they are then required to provide an open-ended explanation. We collect explanations because we want to understand better how subjects reason about choosing a type, both when the type annotation is correct and when it is incorrect.

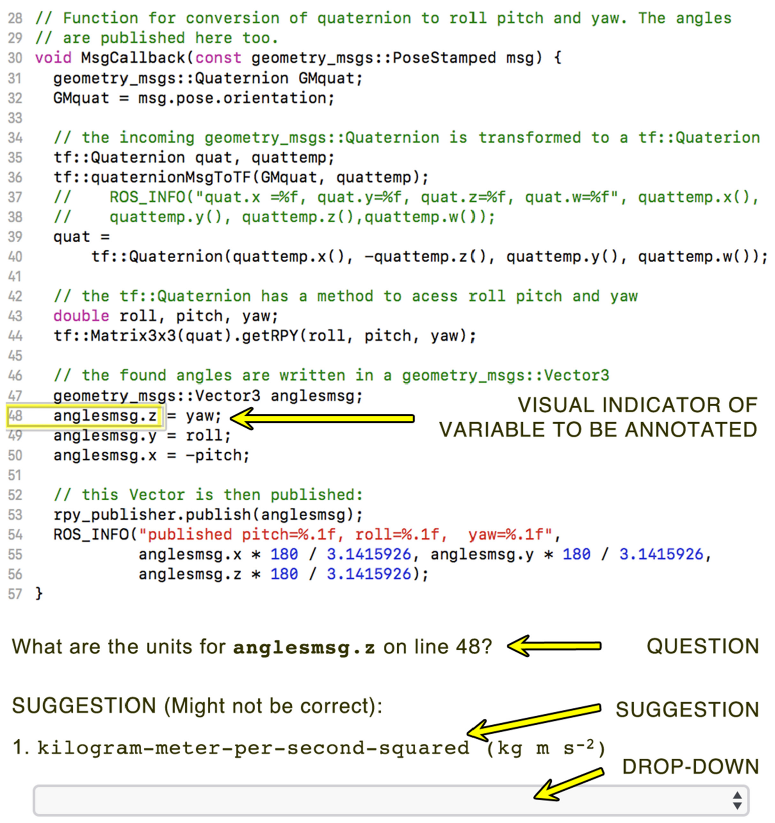
These are the research questions we address in this work. The next section describes the experiment we conduct to address these questions.

3.2 Experimental Setup

In our experiments, we administer a web-based test with questions based on code artifacts. Each test is a collection of 10 questions drawn from a pool of 20 code artifacts. We apply treatments to questions to explore our research questions.

3.2.1 Type Domain and Code Artifacts. There are various type domains, each with specific challenges and characteristics. For all type domains, developers seeking to add a type annotation must reason about the abstract type and choose a type. We choose to instantiate the type annotation task (Section 2.3) within the physical unit types domain, described in Sections 2.1 and 2.2. We choose physical units for several reasons: this domain is generally accessible to anyone with some physics background and includes all software systems that interact with real-world quantities, like robotic and cyber-physical software. Further, subjects will have some previous exposure to physical unit quantities, and we have a particular interest in robot software.

We collected code artifacts for our test instrument from a universe of open-source robotics and cyber-physical code [52] (code that uses "ROS" [59], a popular robotics middleware with many variables representing "real-world types" [81]). As shown in Figure 1, this code is available on GitHub and repositories are selected for inclusion because some file contains a string matching the name of a ROS library, such as "`geometry_msgs::Pose`." When a string matches the name of an ROS library, it is likely that the file containing that string uses the shared data structures defined in those ROS libraries, and therefore that file contains other variables implicitly representing physical quantities that can be annotated with a physical unit type. There are 797,410 `C++` files in the code corpus based on case-insensitive filename suffixes (`.cpp`, `.c++`, `.cxx`, `.cc`) from 3,484 repositories. We narrowed these 797,410 `C++` files to 31,928 first using the tool Phriky [51] to find `C++` files containing physical unit types, and then excluding files that did not parse. From these 31,608 files, we randomly selected functions. We only allowed functions that met the following criteria: (1) no "getter" or "setter" functions, (2) more than 10 lines of code, (3) not just time (*second*) alone



```
28 // Function for conversion of quaternion to roll pitch and yaw. The angles
29 // are published here too.
30 void MsgCallback(const geometry_msgs::PoseStamped msg) {
31   geometry_msgs::Quaternion GMquat;
32   GMquat = msg.pose.orientation;
33
34   // the incoming geometry_msgs::Quaternion is transformed to a tf::Quaterion
35   tf::Quaternion quat, quattemp;
36   tf::quaternionMsgToTF(GMquat, quattemp);
37   //    ROS_INFO("quat.x =%f, quat.y=%f, quat.z=%f, quat.w=%f", quattemp.x(),
38   //    quattemp.y(), quattemp.z(),quattemp.w());
39   quat =
40       tf::Quaternion(quattemp.x(), -quattemp.z(), quattemp.y(), quattemp.w());
41
42   // the tf::Quaternion has a method to acess roll pitch and yaw
43   double roll, pitch, yaw;
44   tf::Matrix3x3(quat).getRPY(roll, pitch, yaw);
45
46   // the found angles are written in a geometry_msgs::Vector3
47   geometry_msgs::Vector3 anglesmsg;
48   anglesmsg.z = yaw;
49   anglesmsg.y = roll;
50   anglesmsg.x = -pitch;
51
52   // this Vector is then published:
53   rpy_publisher.publish(anglesmsg);
54   ROS_INFO("published pitch=%.1f, roll=%.1f,  yaw=%.1f",
55            anglesmsg.x * 180 / 3.1415926, anglesmsg.y * 180 / 3.1415926,
56            anglesmsg.z * 180 / 3.1415926);
57 }
```

What are the units for **anglesmsg.z** on line 48?

SUGGESTION (Might not be correct):

1. kilogram-meter-per-second-squared (kg m s^{-2})

Fig. 2. A code artifact used in the main study. This test question shows treatment T$_3$, an incorrect suggestion. All code artifacts used in this work are available at https://doi.org/10.5281/zenodo.1311901.

because these are often trivially updating a local timestamp, and (4) code that had interactions between different types. We established these criteria to ensure variety, capture interactions in the type domain, and avoid trivially easy artifacts. We repeatedly selected and screened functions until we had 20 artifacts. Within each artifact, we randomly select a variable for type annotation, with at least two of the authors reviewing the selection. All code artifacts used in this study are available at https://doi.org/10.5281/zenodo.1311901, with the drop-down options on the last page.

3.2.2 *Treatments.* Figure 2 shows a sample question and artifact from our test. As shown in the figure, we provide a visual indicator for the variable to be annotated, and this visual indicator corresponds to the line number referenced in the question text below the code artifact. Below the question text, the figure shows an incorrect suggestion (treatment type T$_3$). Other questions might have "No Suggestion" (T$_1$), a correct suggestion (T$_2$), or multiple suggestions (T$_4$–T$_6$). Finally, at the bottom of the question is a drop-down box with several annotation options of physical unit types, discussed shortly below in Section 3.4. As shown in the figure, this code artifact is a callback function in a reactive software system. We seek to approximate the annotation task described in Section 2.3 by asking developers to choose a type for a variable in the code artifact. We measure both the accuracy and time it takes for the developer to select an annotation.

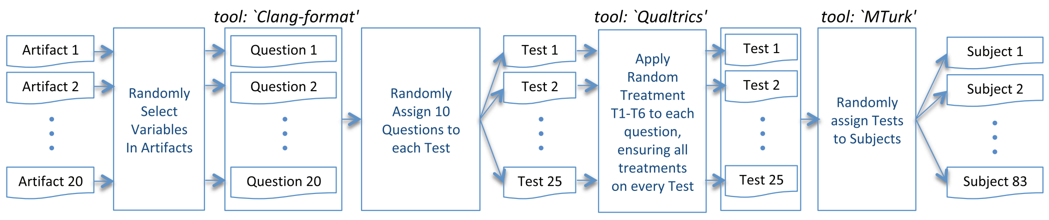

tool: `Clang-format'
tool: `Qualtrics'
tool: `MTurk'
Artifact 1
Artifact 2
Artifact 20
Randomly Select Variables In Artifacts
Question 1
Question 2
Question 20
Randomly Assign 10 Questions to each Test
Test 1
Test 2
Test 25
Apply Random Treatment T1-T6 to each question, ensuring all treatments on every Test
Test 1
Test 2
Test 25
Randomly assign Tests to Subjects
Subject 1
Subject 2
Subject 83

Fig. 3. Experimental design showing how code artifacts become test instruments applied to subjects.

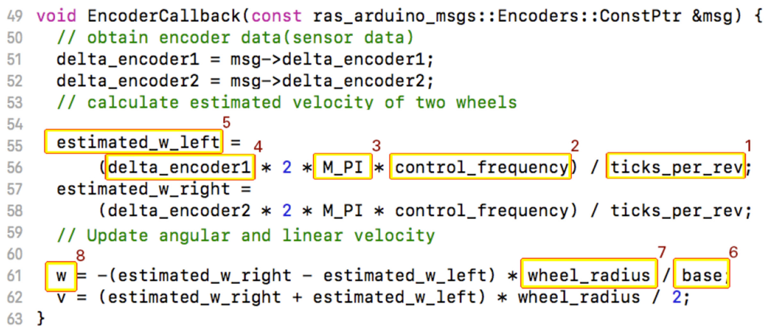
Treatments. To address our research questions, we apply to each question one of six treatments:

- **T_1: "No Suggestion" (control).** A question with the suggestion section not included.
- **T_2: "One Correct."** 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)."* We include this caveat to encourage subjects to approach suggestions with skepticism.
- **T_3: "One Incorrect."** This treatment is identical to T_2 except the suggestion is incorrect. This suggestion is chosen randomly from Table 1 (excluding the correct answer and OTHER).
- **T_4: "Three, 1st Correct."** A question with three suggestions immediately above the drop-down box. The suggestions are each on their own line and are enumerated 1, 2, 3. All suggestions exactly match one option in the drop-down.
- **T_5: "Three, Correct 2nd or 3rd."** This treatment is identical to T_4 except the second or third option exactly matches an option in the drop-down. We randomly placed the correct option either second or third.
- **T_6: "Three, None Correct."** This treatment is identical to T_4 except that *none* of the three suggestions are correct.

Treatment T_1 answers both RQ_1 and RQ_2. T_1 answers RQ_1 by measuring a baseline accuracy and answers RQ_2 by measuring how long annotations take without a suggestion. Treatment T_1 is the control for RQ_1 and establishes a baseline accuracy and timing for our research question about the impact of suggestions. To address RQ_1, we compare the accuracy and timing of questions with treatment T_1 to those treatments with suggestions, T_2–T_6. Also addressing the portion of RQ_1 pertaining to multiple suggestions, we compare the accuracy and timing of questions with "One Correct" (T_2) to "Three, 1st Correct" (T_4), and we compare the accuracy and timing of questions with "One Incorrect" (T_3) to "Three, Correct 2nd or 3rd" (T_5) and "Three, None Correct" (T_6). For the qualitative question, RQ_1, after every question we require subjects to provide an open-ended textual explanation of their reasons for choosing a type. We examine all the explanations utilizing Open Coding [47] to answer RQ_1. Our **independent variable** is the kind of suggestion, if any, and the **dependent variables** are response accuracy and duration.

3.2.3 Experimental Design. As shown in Figure 3, our experimental design is *totally randomized* [36]. In this design, we randomly select a variable to annotate in each of 20 code artifacts, creating 20 questions. We randomly assign subsets of 10 questions from the 20 questions to create 25 tests, balancing the tests so that each question appears nearly the same number of times. We then apply treatments T_1–T_6 randomly to questions in tests, ensuring that each test has every treatment and no more than three of any single treatment. We then apply tests randomly to subjects.

3.2.4 Experimental Design for $\mathbf{RQ_3}$. The experimental design for $\mathbf{RQ_3}$ involves having subjects make multiple type annotations in the same code snippet, as shown in Figure 4. The figure shows



```
void EncoderCallback(const ras_arduino_msgs::Encoders::ConstPtr &msg) {
  // obtain encoder data(sensor data)
  delta_encoder1 = msg->delta_encoder1;
  delta_encoder2 = msg->delta_encoder2;
  // calculate estimated velocity of two wheels

  estimated_w_left =
      (delta_encoder1 * 2 * M_PI * control_frequency) / ticks_per_rev;
  estimated_w_right =
      (delta_encoder2 * 2 * M_PI * control_frequency) / ticks_per_rev;
  // Update angular and linear velocity

  w = -(estimated_w_right - estimated_w_left) * wheel_radius / base;
  v = (estimated_w_right + estimated_w_left) * wheel_radius / 2;
}
```

Fig. 4. A code artifact used for the follow-on study to answer **RQ**$_3$. Number labels indicate the order in which successive type annotations are made by the subject. During the experiment, only one highlighted box is shown to subjects at a time.

the order in which subjects were asked to make multiple type annotations—right to left, top to bottom. Subjects are allowed to go back and revise previous annotations. We chose six artifacts from the pool of 20 (artifacts for questions 3, 4, 6, 15, 19, 20) because they had multiple, related variables that have physical unit types. These variables are related either by transitive assignment or because they are involved in a mathematical expression. We highlight variables one at a time by line number and right to left, asking subjects to associate a type for one variable at a time. These artifacts have between four and eight variables each. As in the main test, we exclude variables that are about time because our subjects nearly always annotate time variables correctly. We show three snippets to each subject. We require explanations as in the main test. We record the time for subjects to make each type annotation in the same way as in the main test.

3.3 Subject Sample Population

We recruited subjects using Mechanical Turk (MTurk), an online marketplace for labor that is popular for many kinds of empirical research [7, 68] including software engineering [14, 42, 70]. MTurk subjects are appropriate for studies requiring neurological diversity [30, 43]. We advertised our study on MTurk as follows:

> *"Academic research pre-screener (≈10 minutes) about computer programming. Prerequisite for $10 task (≈25 minutes). Must have some experience with computer programming. Look at source-code snippet, answer multi-choice questions about computer programming with physical units (like 'meters-per-second,' 'seconds'). Explain your answers."*

Within MTurk, a pretest is known as a "*prescreener.*"

One caveat in collecting demographics on MTurk is that respondents fabricate demographic answers to qualify [32]. Therefore, we clearly state during demographic questions that demographic answers will not be used to determine eligibility and are "NOT GRADED OR SCORED." We warn users to watch for random "attention checks" [27] because the idea of attention checks improves performance, even without enforcement (we do not assess or enforce attention).

We prescreen our subjects using recommended best practices that have been shown not to bias behavioral experiments [73]. The prescreening has two requirements: (1) previously complete at least 500 MTurk tasks with $>$ 90% accuracy and (2) correctly complete a pretest with two physical unit type annotation questions. We pay subjects $2 USD to complete the pretest and $10 USD to complete the main 10-question test. Paying subjects just on correct answers seems like an incentive

Table 2. Demographics for 97 Subjects

YEARS EXPERIENCE	PROGRAMMING C, C++, C#, Java	EMBEDDED SYSTEMS, CYBER-PHYSICAL, ROBOTICS
< 1	26 (27%)	75 (77%)
$1 - 5$	52 (54%)	19 (20%)
5+	19 (19%)	3 (3%)

to provide better answers, but we do not do this because it is ineffective [44]. We encourage subjects not to rush and to provide thoughtful explanations.

We ask three demographic questions during the pretest to try to better understand our subjects' previous experience and to see if these demographics correlate with performance. Table 2 shows a summary of the demographics for our 97 subjects. We ask about experience with (mostly) statically typed languages: *"How many years of programming experience in languages like `C`, `C++`, `C#`, `Java`?"* More than half our subjects (54%, 52/97) report 1 to 5 years' experience with these languages. We then ask about embedded system programming: *"Years of experience programming embedded systems or robotic systems or cyber-physical systems (Things that move or sense)?"* Only 23% (22/97) of subjects report 1 or more years of experience with embedded systems. And third, we inquire about previous experience with code annotation: *"Have you used any code annotation frameworks?"* If subjects report having previous annotation experience, we further ask them to indicate which frameworks they have used. Only 13% (13/97) of subjects indicate experience with annotation frameworks such as "`Resharper/Jetbrains`," "`JSR 308`," and "`SAL/MSDN`." In Section 4.1, we examine the impact of demographics on annotation accuracy.

3.4 Test Instrument Details

Type Annotations Options in the Drop-Down Menu. The drop-down menu includes the 19 physical unit types listed in Table 1, plus `OTHER` and `NO UNITS`. We include `OTHER` to allow subjects to think beyond the options we have provided, and `NO UNITS` captures cases when the variable to be annotated does not belong in the type domain. The `OTHER` option is useful for less common types (i.e., *kilogram-meter-squared-per-second-cubed-per-ampere*, more commonly known as *Volts*, an answer to one of our questions). The `NO UNITS` option is important because the type annotation task first requires developers to identify whether a variable belongs in the domain before selecting a type. The order of the elements in the drop-down menu is randomized every time a subject sees a question. Randomizing the order mitigates the threat of response order bias [46].

Question Timing. We instrument our web test to collect timing information for each question. Our test consists of alternating multiple-choice and open-ended questions. In the multiple-choice question, subjects associate a type (if any) to a variable. Then in the open-ended questions, subjects explain why they selected that type. By tracking how long it takes for subjects to finalize their selection for a type by clicking "next," we can answer RQ_1. We track the time to provide an explanation, but that time is excluded in our results for RQ_1. We do not limit the time to answer individual questions and instead limit the total test duration to 4 hours.

Suggestions. We provide suggestions (as shown in Figure 2) to answer RQ_1 and to assess the impact of future tools that might help developers make type annotations. All suggestions are drawn from the union of the most common unit types in Table 1 along with `NO UNITS` and `OTHER`. The exact suggestion depends on the treatment. Please see Section 3.2.3 for how suggestions are used in treatments. We randomize incorrect suggestions *per test* so that each question-treatment combination receives an assortment of suggestions. Although randomly selected, suggestions are drawn from the 20 most common physical unit types found in a large code corpus, so they are much more likely to be relevant than selecting base units and exponents at random, i.e., *candela-per-kilogram-squared-per-Kelvin-cubed,* which is nonsense.

Explanations. To answer RQ_1, we require subjects to provide textual explanations for why they chose a particular type. We record explanations because we want to better understand the sources of evidence and how that evidence is used. After subjects have finalized their type selection, we again show them the code artifact but with their answer and an open-ended text box. We notify subjects in the instructions that good explanations are required to successfully complete the test.

3.5 Utilized Tools

We use off-the-shelf tools in our experiments and analysis:

Phriky [51] is a static analysis tool to detect physical unit type inconsistencies in ROS `C++` code. We use Phriky to identify `C++` files that contain variables with physical units by invoking it with the `-only-find-files-with-units` command line parameter.

Clang-format [2] is a tool to format `C++` code in a standard way. We use it to ensure that code artifacts shown to subjects are formatted clearly and uniformly.

Qualtrics [3] is a web-based survey tool. We build our test instrument using Qualtrics and use several of its features, such as (1) tracking the time required by subjects to make an annotation, (2) ensuring that all questions are answered, (3) randomizing the question order by subject, (4) randomizing the order of options in the drop-down box for every question, (5) preventing the same IP address from taking the test, (6) recording subjects' responses, and (7) creating unique IDs used to pay subjects. We use Qualtric's API to immediately notify us when a subject passes the pretest so we can grant them access to the main test. We grant access using Mechanical Turk.

Mechanical Turk [1] (MTurk) is a marketplace for online labor. We use MTurk to recruit and pay subjects for both the pretest and main test and retain only anonymized identifiers for remuneration as required by our IRB (# 20170817412EX). We use MTurk to control access to our tests using MTurk's "Qualification" mechanism where we can designate subjects as having passed the pretest as a necessary prerequisite to see the "main test" task.

MySQL. We use a relational database to organize and track tests, questions, suggestions, demographics, and explanations. We use `MySQL` to store data, but to analyze it we use the `R` language.

R Language [60] is a statistical analysis tool that we use to analyze data. We utilized standard packages such as `nnet` for our binomial log-linear response model [78] (`multinom`), the `binom` package [16] for confidence internals, and the `aov` function to perform ANOVA on timing.

3.6 Study Phases

Our study has four phases: a test evaluation phase, a main test deployment phase, a follow-on phase to study multiple suggestions, and a follow-on phase to study multiple related annotations.

Phase One: Evaluation and Refinement of the Test Instrument. During the evaluation and refinement phase, we deploy an initial version of the test to 27 subjects. This initial version has "No Suggestions" (Treatment T_1). This evaluation aims to make sure the questions can be answered correctly by some subjects, are not trivial, and identify areas where our instructions were unclear. We made several iterative improvements to our test instrument based on this initial evaluation: (1) identified two trivial questions and replaced them with more difficult ones; (2) added text to qualify that suggestions *"Might not be correct"*; (3) added to demographic questions that answers

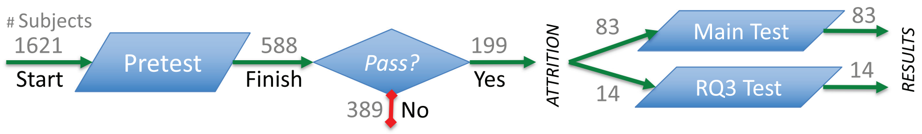


Fig. 5. Number of subjects at each point during Phases Two, Three, and Four combined.

would not be used to screen participants by adding the text *"NOT GRADED OR SCORED,"* in accordance with MTurk best practices [32]; (4) visually identified the variable to be annotated using colorblind-safe yellow markers as shown in Figure 2; (5) ensured that the question order was randomized per subject; and (6) modified the test so that every annotation question was followed by a required, open-ended question about why developers made the annotations they did. The data collected during the evaluation phase is used only for evaluation and refinement, and we excluded all 27 evaluation test subjects from the experiment's deployment phase.

Phase Two: Deployment of Pretest and Main Test. Subjects must pass a pretest and provide good explanations to qualify for our experiment. The pretest serves several purposes. First, it ensures that every subject has some chance to complete the annotation task in the main test and that the explanations will be coherent. Two pretest subjects were screened for providing useless explanations such as "asdf" or "nope" even though they correctly identified the physical unit type. Second, the pretest is a kind of tutorial and includes two practice questions to familiarize subjects with the mechanics of the web test instrument.

Phase Three: Follow-On Survey for Multiple Suggestions. This phase is identical to Phase Two except that more questions had treatments T_4–T_6. We collected these additional responses to answer the portion of RQ_1 that pertains to multiple suggestions with statistical significance.

Phase Four: Follow-On Survey for Multiple Related Annotations. During Phase Four, we conducted a follow-on study to measure the impact of making multiple related type annotations in the same code region. This phase was similar to previous phases, with the same recruitment and pretest process, except that we asked subjects to annotate multiple related variables from the same code artifact. Further, as with the main study, we excluded annotations about time (Section 3.2.1) because they are too easy. We used six artifacts from the set of 20 used in the main study.

Figure 5 shows the number of subjects in Phases Two, Three, and Four combined. As shown in Figure 5, 1,621 subjects started the pretest, but only 588 finished it, indicating that many subjects opted out of the task. Of those that finished the pretest, 34% of subjects (199/588) passed the pretest. We administered the pretests in three 10-hour batches in three days. For the pretest, we gave subjects 30 minutes. After completing the pretest, we reviewed the answers and explanations within 15 minutes of submission (this was possible because we pretested during 10-hour windows over the course of 3 days) and enabled subjects to then immediately take the main test. We found that immediately qualifying passing subjects for the main test noticeably reduced attrition. After passing the pretest, subjects could begin the main test anytime within the next 36 hours and had to complete the main test within 4 hours once started. During Phases Two and Three, we received 833 responses to the main test. During Phase Four, we received 184 responses to the follow-on survey.

4 RESULTS

This section presents the results of the research questions presented in Section 3.1. We describe results for accuracy and time, discuss the impact of suggestions, and finish with qualitative results for how and why developers select particular types.

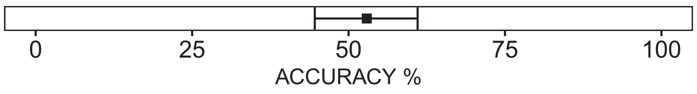


Fig. 6. Manual annotation accuracy for control treatment T_1.

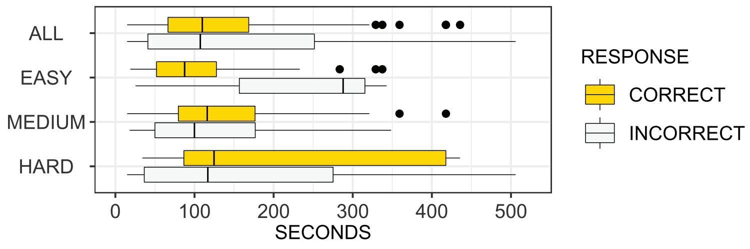


Fig. 7. The time required for a single annotation question under treatment T_1 ("No Suggestion," the control), grouped by question difficulty and correctness. Box whiskers indicate $\pm 1.5 \times IQR$.

4.1 RQ_1 Results: Accuracy

Treatment T_1, "No Suggestion," is the control for our experiments. In T_1, subjects performed the annotation task without suggestions. As shown in Figure 6, the average accuracy for associating unit types to identifiers is 51% (71/138), ±8.5% (Agresti-Coull) [5]. Our results strongly support the commonly held opinion [9, 18] that the type annotation task is difficult without assistance. In practice, accuracy would be improved by many factors, such as training, IDEs, greater context, and increasing familiarity with the code, so our results could provide an upper bound on the difficulty of associating a type. We asked subjects about their previous experience with programming languages, embedded systems, and annotation frameworks, as discussed in Section 3.3. We did not find significant differences based on their reported demographics.

Incorrect Answers. For incorrect answers, the most common mistake for all questions was `NO UNITS`. This is interesting because it means that not only do developers need help determining the correct type annotation but also *developers need help determining which variables need a type annotation.*

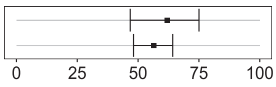
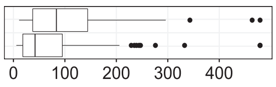
RQ_1 Results: Manually associating type annotations is error-prone (51% accurate, ±8.5%).

4.2 RQ_2 Results: Timing

Using the accuracy of responses with the control treatment T_1, we group questions into three groups by difficulty. The groups are Easy [100% – 75% correct], Medium [75% – 25%], and Hard [25% – 0%]. We grouped questions this way to explore how difficulty correlates to other aspects, like timing and the impact of suggestions. We tried several groupings (two through five groups) and found they all exhibit similar facets of the same underlying contours. Detailed accuracy and timing results for questions arranged by difficulty and treatment are shown in Appendix A, Table 7, including the response accuracy (percentage and fraction) and timing (mean and median). The table shows results by question, and the code artifacts corresponding to each question are available at https://doi.org/10.5281/zenodo.1311901, with the drop-down options on the last page.

As shown in Figure 7, making a single correct annotation in previously unseen code takes 135s (median = 110s), with some outliers capped. Our timing data contains outliers, perhaps because we allowed subjects 4 hours to complete the test and administered the test via the web and therefore

Table 3. **RQ$_3$**: Accuracy and Timing for Making Multiple Annotations in the Same Code Region

ANNOTATION	ACCURACY (%)	TIMING (seconds)	RESPONSES	SUBJECTS
First			42	14
Subsequent			142	14

could not observe how subjects spent their time. To address the furthest outliers, we Winsorize [26] outliers to the 2nd/98th percentiles.

The time required to make a correct annotation measured by our experiment does not include the additional time required to determine what variables do not belong to the type domain, troubleshoot incorrect annotations, or maintain type annotations during evolution.

RQ$_2$ Timing Results: The type annotation task is time intensive (mean = 135s, median = 110s for a single variable).

4.3 RQ$_3$ Multiple Related Annotations

The results of **RQ$_3$** are based on a follow-on study with 14 subjects and 184 responses. In this follow-on study, we used 6 of the 20 artifacts from the main test. We showed subjects three code snippets and asked them to provide physical type annotations for between four and eight variables in the same snippet. See Section 3.2.4 for more details.

Our null hypothesis is that the accuracy for the first annotation is the same as the accuracy for subsequent annotations. We designed this experiment to be sensitive to a change in accuracy of ±10% by requiring a sufficient number of subjects and responses to achieve a 95% binomial confidence interval (Agresti-Coull [5]). As shown in Table 3, the accuracy of this first annotation in this experiment was 53%, nearly identical to our results of 51% for **RQ$_1$** (Section 4.1). Subsequent annotations had an accuracy of 58%, not a large enough difference for us to reject the null hypothesis. From this, we conclude that developers do not become significantly more accurate by making multiple related annotations in the same code region.

The first annotation's duration was 119 s, similar to our baseline average of 135 s, but subsequent annotations were significantly faster than the baseline ($p < 0.05$). We detected this speedup between the first and second annotations, with all annotations after the first taking an average of 79 s (66% of the time of the first iteration). We conjecture that once a developer has invested the time to analyze the code to assign the first annotation, part of that fixed cost can be leveraged to reduce the Type Annotation Burden (Definition 2.1). We note, however, that the annotation does not get noticeably faster even after the second annotation.

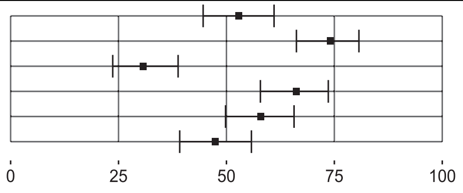
RQ$_3$ Multiple Related Annotations: Making several related type annotations does not increase accuracy significantly but increases speed.

4.4 RQ$_4$ Results: Impact of Suggestions on Accuracy

The results of **RQ$_4$** are based on the main study with 83 subjects and 833 responses.

Our results for accuracy are based on responses to a multiple-choice question where the answer can either be correct or incorrect, a binomial outcome. Because we want to measure how treatments (suggestions) impact accuracy, we need a mathematical model to quantify the impact. We use a binomial log-linear response model [78] because it helps us measure significance between

Table 4. Annotation Accuracy by Question Treatment

TREATMENT (SUGGESTION)	ACCURACY (%)	RESPONSES	SUBJECTS
T_1 No Suggestion (control)		140	72
T_2 One Correct		139	69
T_3 One Incorrect		142	58
T_4 Three, 1st Correct		136	66
T_5 Three, Correct 2nd or 3rd		146	69
T_6 Three, None Correct		141	69

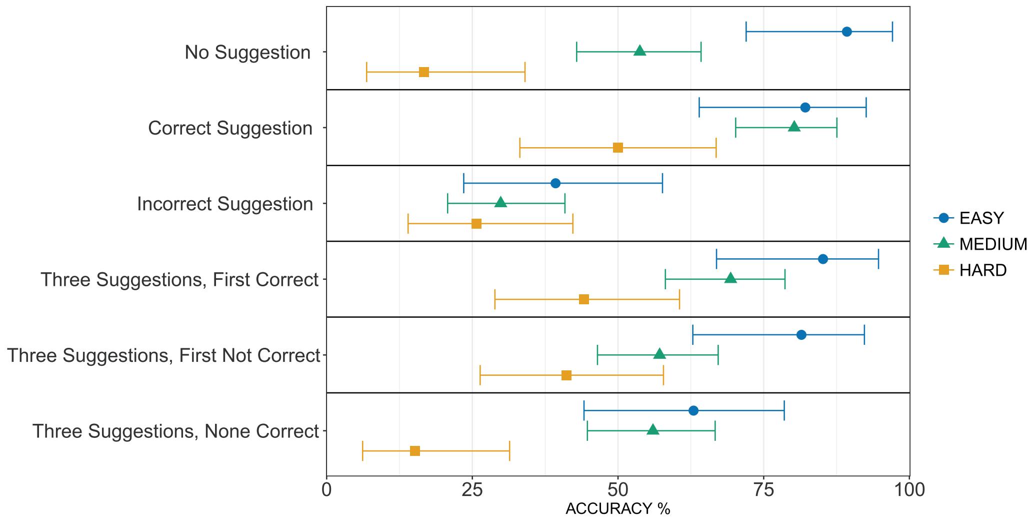


Fig. 8. Annotation accuracy per treatment and question difficulty. Intervals indicate 95% confidence levels.

treatments and visualize the uncertainty. The impact of suggestions results in a type annotation that is either *correct*= 1 or *incorrect*= 0.

4.4.1 Suggestions Have a Significant Impact on Accuracy. As shown in Table 4, single suggestions have strong and significant effects on annotation accuracy. "One Correct" suggestion (T_2) improves accuracy by 21% compared to "No Suggestion" (T_1) ($p < 0.001$). "One Incorrect" suggestion (T_3) decreases accuracy by 23% compared to "No Suggestion" (T_1) ($p < 0.0001$).

Figure 8 shows the range of accuracy for all treatments by question difficulty. "One Correct" suggestion (T_2) benefits all questions compared to "No Suggestion" T_1, with similar improvements for HARD (+33%) and MEDIUM (+26%) questions, while only helping EASY questions by +7%. As shown in the figure, "One Incorrect" suggestion T_3 reduces accuracy for EASY (−53%) and MEDIUM (−49%) questions with little impact on HARD questions. This makes sense because subjects who could correctly determine the correct type annotation for a HARD question already had evidence that eliminated the incorrect suggestion.

In our randomized experimental design, some subjects ($N = 39$) saw an incorrect suggestion first, while others ($N = 44$) saw a correct suggestion first. We wondered if this could cause an anchoring bias [75] where a subject's first encounter with the correctness of suggestions would bias their willingness to accept later suggestions. We examined and compared later responses from these two groups and found no significant difference between their likelihood to answer

Table 5. Summary of Type Annotation Explanations for 833 Answers

	CORRECT RESPONSES						INCORRECT RESPONSES						TOTAL	
EXPLANATION CATEGORY	T_1	T_2	T_3	T_4	T_5	T_6	T_1	T_2	T_3	T_4	T_5	T_6	#	%
Names only	36	54	17	40	36	27	35	20	44	23	21	20	373	48%
Math reasoning and names	20	24	18	26	32	23	5	4	12	9	10	19	202	26%
Not in type domain	4	10	1	7	8	6	19	13	25	4	18	18	133	17%
Code comments	11	9	2	5	6	5	3	-	-	-	2	-	43	5%
Used suggestion	-	5	-	2	1	-	-	-	12	-	1	-	21	3%
Type depends on input	-	-	-	-	-	-	5	2	2	-	1	1	11	1%

with a suggested answer. This is important because it suggests that our methodology of drawing suggestions randomly from a pool, which causes some suggestions to be "closer" than others, does not bias subjects' adoption of suggestions in later questions much.

4.4.2 Multiple Suggestions Are Better Than Single Suggestions. As shown in Table 4, "One Correct" suggestion (T_2) has a similar distribution of accuracy to "Three, 1st Correct" (T_4). Therefore, showing three suggestions is nearly as helpful as showing a single suggestion. Also notice that "Three, None Correct" T_6 causes significantly less harm than "One Incorrect" T_3 for EASY and MEDIUM difficulty questions. Subjects were significantly more likely to "take the bait" when presented with a single incorrect suggestion (30/98) than when presented with three incorrect suggestions (17/147). Since making multiple suggestions helps nearly as much as a single correct suggestion, and making multiple suggestions hurts significantly less than a single incorrect suggestion, we assert that multiple suggestions are better than a single suggestion.

4.4.3 Suggestions Do Not Significantly Slow the Type Annotation Task. Overall, we found no significant difference between response times, even when considering treatment or question difficulty. The significant benefits of making a correct suggestion do not cause a time penalty, even when three suggestions are wrong.

RQ_4 Suggestion Results:

(1) Single suggestions have a significant impact on developers' accuracy.
(2) Multiple suggestions help developers as much as single suggestions but without the negative effect when the suggestions are incorrect.
(3) Suggestions do not significantly affect developers' time spent annotating types.

4.5 RQ_5 Qualitative Results: Clues for Choosing a Type

Annotation explanations are *qualitative*, unlike the results from Sections 4.1 through 4.4.3 that are *quantitative*. To better understand how and why developers choose a type annotation, we explore the explanations subjects provided after each annotation using Open Coding [47]. In Open Coding, the goal is to categorize explanation elements into cohesive groups where each group represents a single concept. In this approach, rather than start with predefined labels, we instead examine each explanation successively in random order and assign and create labels simultaneously so that categories emerge from the data organically. The process is iterative, and during the first iteration, we identified 12 categories. With each successive iteration, we merged labels until, after three iterations, we converged to six labels. Table 5 shows the six labels. Explanations might receive multiple labels, such as a variable name reinforced by a comment. The table shows the labels we identified broken down by treatment and whether the annotation was correct.

The most common explanation was "Names only," providing almost half (48%) of all responses. The importance of high-quality identifiers is well supported [11], and our results confirm that identifiers are a significant factor in how subjects make the semantic connection between variables and their types, for example:

> *"The name of the left part of the expression is msg.linear_acceleration.z. I trust the [person] who coded this and thus I think that this would be in units of linear acceleration (meters per second squared)."* —Response to Q^{13}

> *"At least I hope 'torque' is referring to torque."* —Response to Q^{17}

Note that "Names only" is also the most common explanation for incorrect type annotations, indicating that the clues in variable names can be misleading, confusing, or insufficient. Our results regarding the importance of variable names should be taken with a grain of salt, however, because every code artifact in our study had some identifier, but not all artifacts had comments or mathematical reasoning.

The second most common explanation is "Math reasoning and names," accounting for 26% of explanations, such as:

> *"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."* —Response to Q_4

As shown in Table 5, subjects providing incorrect answers are less likely to identify "Math reasoning and names" as their reason for choosing a type. However, some subjects cite math reasoning and then bungle the math:

> *"Meters per second times dt would cause the seconds to cancel out and the meters to square,"* —Response to Q_4

where *"cause...the meters to square"* is not correct.

"Not in type domain" is the third most common explanation provided for choosing a type (17% overall), but 73% (97/133) of these responses are incorrect. "Not in type domain" means that the identifier should not be associated with a type because there is no corresponding type in the physical units type domain. It appears that subjects were unable to see how the variable in question belonged to the type domain. This raises an important question in the overall process of associating types: which variables should be typed? It appears that future tool developers could aid the type annotation task just by helping find these variables.

The fourth most commonly cited reason is "Code comments" (5%), with comments much more likely to be cited with correct answers ($N = 36$) than incorrect answers ($N = 5$). Note that only 2/20 of our code artifacts contained comments (Q_6 and Q_8). This might indicate that although code comments are effective in providing a clue to the correct type annotation, the overall lack of comments limits this as a factor.

Only 3% of explanations (21/833) explicitly say they just took the provided suggestion. However, this is likely only a lower bound because 63/463 (14%) of incorrect responses matched an incorrect suggestion, and subjects might not have admitted to using the suggestion.

Qualitative Results: The main clues for type selection are identifier names and reasoning over code operations. A useful future tool would suggest better identifiers.

5 LIMITATIONS

Subjects Not Familiar with the Code. This study considers a code evolution scenario and therefore we asked subjects to annotate code they did not write. Applying type annotations to someone else's previously existing code can happen when developers seek to improve overall code quality by gradually evolving code into typed code [61, 72].

Subjects Not Familiar with Annotation Task. Subjects were not trained on the annotation task, and we do not assume any background with type annotations. Since subjects were not specifically trained to annotate physical unit types, our results likely underestimated trained developers' true accuracy. Training subjects to perform this task could improve accuracy, but we wanted to establish that the basic annotation task is not trivially easy.

Subjects Not Representative of Developers. Our subjects are recruited from Mechanical Turk and might not represent developers more generally, even if MTurk is appropriate for research seeking neurological diversity [30, 43]. We mitigate this threat by requiring subjects to correctly annotate two code artifacts during the pretest and provide good explanations for choosing a type. To answer the pretest questions correctly, subjects must comprehend code, understand the annotation task, and correctly identify the physical unit type.

Uncommon Names for Physical Unit Types. Some of the physical unit types in our study have common names, such as the type *kilogram-meters-per-second-squared* being more commonly known as *Newtons*, a measurement of *force*. In our study, we use the fully explicit, long form of the name. To help mitigate the case when subjects do not connect the full name with the common name, we examine every explanation, and when subjects indicate the common name in the explanation and select "OTHER" as an answer, we consider the answer to be correct. Overall, we deemed 7/414 incorrect answers as correct because of an explanation that correctly identifies the common name.

Fidelity of Timing Data. During the time window to take the main test (4 hours), subjects might take breaks or perform other tasks. Since our test instrument is web based and remotely administered, we cannot distinguish between long interludes spent thinking about questions from other activities. This long duration might give rise to "ceiling effects," with longer overall time estimates to complete tasks. We mitigate this threat by identifying and capping some timing outliers (described in Section 4.2). Also, our timing only captures the time to choose an annotation, whereas we are aware that a developer might spend additional time troubleshooting incorrect annotations.

Generalization to the General Type Annotation Task. We concretize this work in the domain of physical unit types as described in Sections 2.1 and 2.2, and this type domain might not generalize to the general type annotation task. We note developers must still reason about interactions in the type system and how code operations impact types no matter the type domain. We ask subjects to associate types for local variables, whereas some type systems like `Hack` [17] only allow type annotations on function signatures. Although reasoning about the types involved in function signatures requires non-local reasoning, in both cases developers must reason about code operations and the type domain. Additionally, we deliver the annotation task through a web-based test, which might not represent how developers apply annotations in an IDE, which might provide access to headers, APIs, filenames, and the ability to guess-and-check a type [77].

Code Artifacts: Generality and Limited Code Context. The code artifacts we selected might not generalize to all code that needs type annotation. We mitigate this threat by randomly selecting code artifacts from a large corpus of 5.9*M* LOC. We use 20 code snippets and note that other software engineering research with code snippets uses between 6 and 23 snippets [15, 38, 66, 67].

Table 6. Correct Types for Each Question Compared to PHRIKY and PHYS Unit Annotations

				SUGGESTIONS								
				PHRIKY	PHYS							
	Q#	VARIABLE NAME	CORRECT TYPE		1st	2nd	3rd	Good Identifiers	Truncated	Multi-Line	LOC	Variable Count
EASY	12	`pose.orientation`	q	✓	✓			✓			18	6
	9	`delta_d`	m	✓	✓			✓	✓		36	3
	5	`robotSpeed.angular.z`	$\mathrm{rad\,s^{-1}}$	✗	✓			✓			10	3
	15	`x2`	$\mathrm{m^2}$						✓	✓	22	4
MEDIUM	4	`delta_x`	m		✓			✓	✓	✓	27	2
	6	`w`	$\mathrm{rad\,s^{-1}}$					✓		✓	14	13
	16	`av`	$\mathrm{rad\,s^{-1}}$					✓			11	3
	8	`path_move_tol_`	m		✓			✓			18	2
	2	`springConstant`	$\mathrm{kg\,s^{-2}}$		✗	✗	✗				7	2
	3	`ratio_to_consume`	NO UNITS		✗	✗	✗	✓		✓	36	12
	7	`x`	NO UNITS						✓	✓	29	4
	10	`wrench_out.wrench.force.y`	$\mathrm{kg\,m\,s^{-2}}$	✓	✓					✓	36	4
	11	`data->gyro_z;`	$\mathrm{m\,s^{-2}}$	✓	✓			✓	✓		27	2
	14	`xi`	m		✓			✓	✓		12	5
	18	`motor_.voltage[1]`	$\mathrm{kg\,m\,s^{-3}\,A^{-1}}$	✗	✗	✓		✓	✓		20	6
HARD	1	`return`	m							✓	10	3
	13	`angular_velocity_covariance`	$\mathrm{rad\,s^{-2}}$	✗	✓				✓		23	2
	17	`torque`	$\mathrm{kg\,m^2\,s^{-2}}$	✓	✓			✓	✓		8	5
	19	`anglesmsg.z`	rad	✓	✓			✓		✓	27	5
	20	`dyaw`	rad	✗	✗	✓			✓	✓	27	18

The test instrument is available at https://doi.org/10.5281/zenodo.1311901, with the drop-down options on the last page.

The context we show to subjects is limited to a function, whereas additional context might help determine the correct type. We mitigate this threat by testing our questions during an evaluation phase (see Section 3.6) to ensure that it is possible to infer the correct units with the available information.

6 DISCUSSION

6.1 Code Attributes' Impact on Annotation Accuracy

We examine several code attributes to determine if these features could make type annotations more difficult. We identified five attributes: "has good identifiers," "is truncated," "requires multi-line reasoning," LOC shown to the user in the code artifact (possibly truncated), and the number of variables involved. The first three are binomial (either True or False), and the last two are integer-valued discrete quantities. Table 6 shows the attributes for each question in our test instrument. We now discuss each in more detail.

"Has Good Identifiers." For this code attribute, we examined each identifier transitively connected to the variable to be annotated. For each of these identifiers, we determined whether the identifiers speak to the type domain or contain substrings that contribute a useful clue to the type. Based on the presence of these semantic clues, we designate the question as having "good

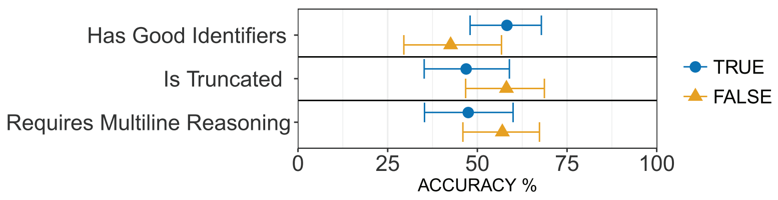


Fig. 9. Code attributes' impact on annotation accuracy for questions with "No Suggestion" (T_1).

identifiers." For example, the code artifact in Figure 2 is designated "good" because `anglemsg` contains the substring "angle" and the right-hand-side `yaw` is semantically connected to angles. These clues together narrow the search even without considering code operations. Conversely, some identifiers contain scant semantic meaning, such as `x2`, `pt`, `k`, and `av`, and others like `tmp_point_out.point.x` have semantic meaning but are misleading in the artifact context (Q_{10}), which is about *force*. We found "Good identifiers" in 13/20 questions.

"Is Truncated." For this attribute, we noted whether the artifact shown to subjects in the main test (Section 3.6) is truncated. We truncated some artifacts because they were significantly longer than what could fit on a screen of our web-based survey instrument. At most we showed 36 LOC. Artifacts are truncated in 10/20 questions.

"Requires Multi-Line Reasoning." For this attribute, we noted whether the operations impacting the type to be annotated are contained within a single line. This single line also has a visual indicator in the artifact (see Section 3.2).

The artifact shown in Figure 2 requires multi-line reasoning because only by considering the code operations on lines 55 and 56 can the type on line 48 be determined to be *radians* and not *degrees*_360. Multi-line reasoning is required for 9/20 questions.

LOC in Artifact. For this attribute, we count the number of non-blank, non-comment lines of code in each artifact using `CLOC` [12]. Unlike the previous code attributes, this attribute is not binomial but integer-valued. The 20 artifacts have a 20 LOC on average ($\sigma = 10.0$).

Number of Variables Involved with Type Annotation. For this attribute, we compute the number of variables that participate in the reasoning by starting with the variable to be annotated and counting all variables in the backwards data dependence slice within the code artifact. We do this because the number of transitively involved variables might correlate with the complexity of the reasoning, and the more ways that reasoning can go wrong. The 20 artifacts have an average of 4.8 variables involved ($\sigma = 4.3$).

6.1.1 Code Attributes Results. Figure 9 shows how some binomial (True or False) code attributes impact accuracy for responses to questions with "No Suggestion" (T_1). We conducted this analysis *a posteriori* to explore how these code attributes impact accuracy. As shown in Figure 9, "Has Good Identifiers" does not reveal a significant effect on accuracy, although identifier names are the most important reason subjects gave for choosing a type (see Section 4.5). Likewise, truncated artifacts do not have a significant impact ($p = 0.18$). We did not measure a significant impact for "Requires Multiline Reasoning" ($p = 0.26$), which we found surprising because of anecdotal experiences with particularly challenging type annotations that required multi-line reasoning. We save further refinement of the role of context in type annotation accuracy for future work.

Figure 10 shows the negligible impact of increasing lines of code in the code artifact. However, as shown in Figure 11, the number of variables involved shows a negative correlation with accuracy, indicating the difficulty developers face when annotating a variable that depends on the interplay

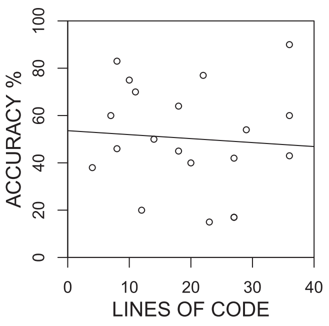


Fig. 10. Accuracy by lines of code in the artifacts.

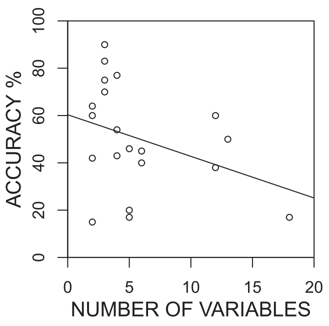


Fig. 11. Accuracy by the number of variables involved.

of several elements of the type domain. Note that we only have 20 examples and that having a small number of instances with larger variables might bias the correlation. As shown in the figure, the more variables involved in an annotation, the more difficult it is to associate a type correctly. This might be a way to rank variables needing annotation by difficulty.

All of these code attributes likely require further, larger-scale studies to definitively characterize their impact on accuracy.

6.2 Assessment of State-of-the-Art Tools to Suggest Unit Types

As shown in Section 4.4, suggestions have a significant impact on a developer's ability to make type annotations correctly. To better understand the capabilities of current tools to make type suggestions in the physical units type domain, we consider two open-source tools: Phriky [51] (version 1.1.0) and Phys [34] (version 1.0.0). Both Phriky and Phys analyze `C++` code that builds against the Robot Operating System (`ROS`). Moreover, Phys makes multiple suggestions, ordered by confidence, of which we consider only the top three to be consistent with our study. We chose these tools because they are state of the art and open source, work on `ROS C++` files, and infer physical unit types.

Table 6 shows the type predictions for the 20 variables in our test instrument made by Phriky and Phys. The tool Phriky makes predictions for 10/20 (50%) of the variables, but only 6/10 (60%) suggestions are correct, whereas Phys makes predictions for 15/20 (75%) variables and gets 11/15 (73%) correct as the first guess (highest confidence), 13/15 (87%) correct in the first or second guess, and 2/15 (13%) completely wrong. Although these tools are a good initial step, tool developers still have opportunities to overcome remaining challenges.

In general, these tools make errors for the following reasons: failing to account for the surrounding context, determining that a variable belongs to the type domain when it does not, and missing domain-specific nuances in the identifiers. As shown in Table 6, Phriky guesses incorrectly about `robotSpeed.angular.z`, which from the surrounding context in Q_5 is about *angular* rotation and not *linear* velocity as Phriky supposed. Some variables, like `ratio_to_consume`, are not in the type domain (Table 6, Q_3), but Phys believes it is (it has no units). Further, some variable names like `w` (Q_6) seem to have very little semantic information, but within the robotics software domain, `w` is used to represent the similar-looking ω (omega), which often means *angular velocity* [74]. These kinds of type inference tools are discussed further in related work (Section 7).

Integrating the suggestions from tools like Phriky and Phys with user knowledge to answer many easy type annotations with high precision could help developers build an early mental model and reduce early annotation costs.

7 RELATED WORK

Empirical Evaluation of Type Systems. Type systems have been shown in many efforts to be beneficial in several aspects, including understandability [71], usability [45], and maintainability [24, 37]. These efforts sought to measure the benefits of using a type system, while our work focuses on the cost of making those annotations. Spize and Hanenberg [69] found that type names, even without a type enforcement mechanism, improved API usability. We also find that variable names contain useful hints (Section 4.5) and are the most common factor for determining a type annotation. Prechelt and Tichy [58] measured how static type checking reduces student programmers' likelihood of introducing defects in `ANSI-C`. We are also interested in measuring the impact of types, but we use code artifacts found in a code corpus rather than creating new artifacts.

Type Annotations. Xiang et al. [81] proposed "Real-World Types" as a type domain to apply type checking to previously untyped code. They demonstrate their approach by examining open-source systems, including a flight planner, and found six previously undetected faults. Their workflow uses a skilled developer to apply type annotations to the code under examination. We likewise are interested in enabling additional code checks with type annotations, but our work looks at the effort of making type annotations. Gao et al. [19] study how many additional bugs can be detected in `JavaScript` code by applying type annotations with `Flow` and `TypeScript` annotation frameworks. In this work, the authors applied the type annotations. Their work identified a *time tax* and *token tax* for each bug to quantify the effort involved in the type annotation task and measured their efforts to detect each bug. From their token and time tax, we infer that their average time to make a single annotation was 136 s for `Flow` and 129 s for `TypeScript`. These durations are preternaturally similar to our measurement of 136 s for a single type annotation, especially since the skill level, type domain, task, and language are different. Both their work and ours seek to better understand the effort in making type annotations, but we focus on measuring a population of 83 subjects rather than using the author's efforts.

Type Inference. There is increasing interest in inferring types automatically using uncertain sources of information. Hellendoorn et al. [28] used deep networks to explore the effectiveness of learning type patterns from `TypeScript` code (type annotated `JavaScript`). Luo et al. [40] inferred types in `Java` from type docstrings, and Dash et al. [13] proposed a tool RefiNym that uses variable names and dataflow to identify variables whose types can be refined. Kate et al. [34] infer physical unit type for `ROS` code with the tool Phys used in Section 6.2. Like these works, we are interested in activating type checking's benefits in untyped contexts using uncertain information (i.e., suggestions). However, unlike these works, we seek to quantify the impact of making suggestions on overall type annotation accuracy and timing.

Type Annotation Assistants and Automatic Suggestions. Nimmer and Ernst [49] examined what factors improved the utility of an annotation assistant. They also conduct an empirical study but focus on assertions' effectiveness, whereas we focus on the effort involved in making type annotations. Vakilian et al. [77] empirically evaluated their type qualifier tool Cascade and found the best results when the tool and developer worked together. We likewise consider annotation assistants that make suggestions to developers to be a promising direction, especially when combining uncertain evidence. Parnin and Orso's [55] work on fault isolation showed that when tools present multiple suggestions to a developer, he or she is likely to use only the first few suggestions. Based on this evidence, we constrain the number of suggestions in treatments T_2–T_6 to three maximum.

Physical Units in Programs. Many, many efforts examine how to apply type checking of physical unit types [4, 10, 21–23, 25, 29, 31, 33, 35, 41, 54, 56, 64, 65, 76, 80]. Nearly all these works require type annotations. We likewise instantiate our efforts in the physical units domain, but unlike these works, we consider the annotation effort required to make the type annotations themselves.

8 CONCLUSION

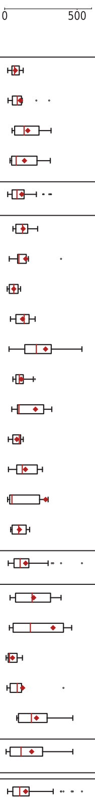
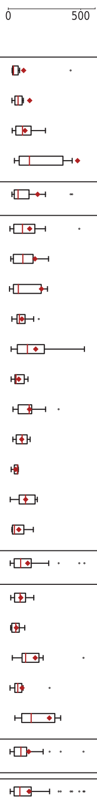
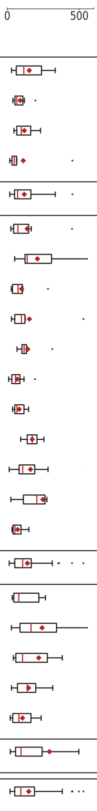
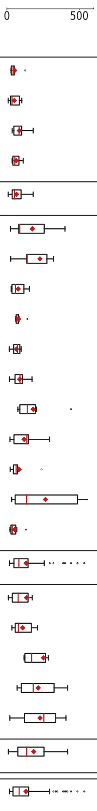
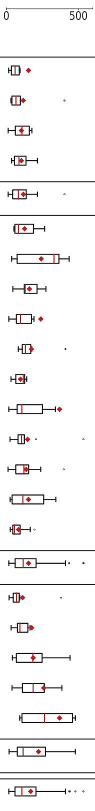
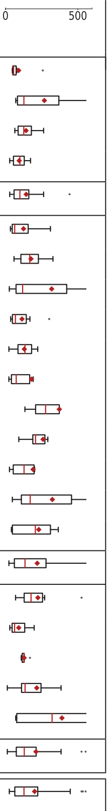
This work makes several contributions to our understanding of the process of making type annotations. We measure manual type annotation accuracy and find that developers choose the correct type only half of the time. We find that making a correct type annotation takes nearly 2 minutes. We further determine that suggestions have a strong impact on accuracy. Because manually making correct type annotations is time intensive and inaccurate, and because suggestions can be very beneficial, our findings indicate significant potential for automated tools to help lessen the Type Annotation Burden (Definition 2.1). Automated tools should make three suggestions instead of one because multiple suggestions are less detrimental when wrong yet nearly as beneficial.

Our qualitative analysis gives hints as to the information developers use when choosing a type and find that identifier names and understanding how code operations impact the type domain are critical factors. Further, current tools that make type suggestions only address a portion of the annotation space. We show that current annotation tools have promise but significant room for improvement. Overall, developers making type annotations could benefit significantly from accurate type suggestion tools.

APPENDIX

A DETAILED ACCURACY AND TIMING STATISTICS

Table 7. Accuracy and Time for Questions by Treatment

Q#	Difficulty	CONTROL			TREATMENTS														
					ONE SUGGESTION						THREE SUGGESTIONS								
		T1			T2			T3			T4			T5			T6		
		Correct		Time (s)	Correct		Time (s)	Correct		Time (s)	Correct		Time (s)	Correct		Time (s)	Correct		Time (s)
		%	Fraction	0 500 	%	Fraction	0 500 	%	Fraction	0 500 	%	Fraction	0 500 	%	Fraction	0 500 	%	Fraction	0 500 
12	Easy	100	6/6		83	5/6		33	2/6		100	6/6		83	5/6		67	4/6	
9		90	9/10		80	8/10		67	4/6		100	6/6		100	6/6		83	5/6	
15		83	5/6		83	5/6		40	4/10		78	7/9		89	8/9		33	2/6	
5		83	5/6		83	5/6		17	1/6		67	4/6		50	3/6		67	6/9	
All Easy		89	25/28		82	23/28		36	10/28		74	23/27		81	22/27		63	17/27	
6	Medium	67	4/6		75	6/8		50	3/6		89	8/9		33	2/6		44	4/9	
4		67	4/6		80	8/10		20	2/10		50	3/6		67	6/9		33	2/6	
16		67	4/6		90	9/10		33	2/6		67	4/6		89	8/9		70	7/10	
8		64	7/11		90	9/10		33	2/6		67	4/6		67	4/6		67	4/6	
3		60	6/10		83	5/6		17	1/6		67	4/6		100	6/6		83	5/6	
2		60	6/10		33	2/6		20	2/10		50	3/6		33	2/6		50	3/6	
7		50	3/6		80	8/10		17	1/6		67	4/6		33	3/9		20	1/5	
10		43	3/7		83	5/6		33	2/6		89	8/9		44	4/9		83	5/6	
11		33	2/6		100	3/3		67	6/9		50	3/6		44	4/9		83	5/6	
18		33	2/6		100	6/6		33	2/6		67	6/9		67	4/6		83	5/6	
14		33	2/6		67	4/6		0	0/6		83	5/6		56	5/9		11	1/9	
All Medium		51	41/80		77	65/84		28	21/74		69	52/75		57	48/84		56	42/75	
19	Hard	17	1/6		50	3/6		17	1/6		67	6/9		33	2/6		50	3/6	
1		17	1/6		67	4/6		40	4/10		67	4/6		33	2/6		17	1/6	
17		17	1/6		33	2/6		57	4/7		67	4/6		43	3/7		0	0/6	
13		17	1/6		50	3/6		0	0/6		0	0/6		56	5/9		11	1/9	
20		17	1/6		50	3/6		0	0/6		14	1/7		33	2/6		0	0/6	
All Hard		17	5/30		50	15/30		23	8/35		44	15/34		41	14/34		15	5/33	
All Questions		53	73/138		74	103/139		31	43/140		66	90/136		58	84/145		47	64/135	

ACKNOWLEDGMENTS

Thanks to our subjects for participating in the study. Also thanks to Urja Acharya, Carl Hildebrandt, Ajay Shankar, and Adam Plowcha and the NIMBUS Lab for providing feedback on early versions of the test. Thanks to Aimee Allard for help typesetting this manuscript.

REFERENCES

[1] Kevin Crowston. 2012. Amazon mechanical turk: {A} research tool for organizations and information systems scholars. In *Proceedings of the Shaping the Future of {ICT} Research. Methods and Approaches - {IFIP} {WG} 8.2, Working Conference*, Anol Bhattacherjee and Brian Fitzgerald (Eds.). IFIP Advances in Information and Communication Technology, Vol. 389. Springer, 210–221. DOI: 10.1007/978-3-642-35142-6_14

[2] 2018. Clang: A C language family frontend for LLVM. https://clang.llvm.org (accessed May 1, 2019).

[3] Chris Lattner and Vikram S. Adve. 2004. The {LLVM} compiler framework and infrastructure tutorial. In *Languages and Compilers for High Performance Computing, 17th International Workshop (LCPC'04)*, Rudolf Eigenmann, Zhiyuan Li and Samuel P. Midkiff (Eds.). Lecture Notes in Computer Science, Vol. 3602. Springer, 15–16. DOI: 10.1007/11532378_2

[4] Mukul Babu Agrawal and Vijay Kumar Garg. 1984. Dimensional analysis in Pascal. *SIGPLAN Notices* 19, 3 (March 1984), 7–11. DOI: https://doi.org/10.1145/948576.948577

[5] Alan Agresti and Brent A. Coull. 1998. Approximate is better than exact for interval estimation of binomial proportions. *American Statistician* 52, 2 (1998), 119–126.

[6] BIPM. 2006. *Le Système International d'unitaés / The International System of Units ("The SI Brochure")* (8th ed.). Bureau international des poids et mesures. Retrieved from http://www.bipm.org/en/si/si_brochure/.

[7] John Bohannon. 2016. Mechanical Turk upends social sciences. *Science* 352, 6291 (2016), 1263–1264. DOI: 10.1126/science.352.6291.1263

[8] Luca Cardelli. 1996. Type systems. *ACM Computuing Surveys* 28, 1 (1996), 263–264. DOI: https://doi.org/10.1145/234313.234418

[9] Patrice Chalin and Perry R. James. 2007. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Object-Oriented Programming, 21st European, Proceedings (ECOOP'07)*. 227–247. DOI: https://doi.org/10.1007/978-3-540-73589-2_12

[10] Robert F. Cmelik and Narain H. Gehani. 1988. Dimensional analysis with C++. *IEEE Software* 5, 3 (1988), 21–27.

[11] 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*. 57–67. DOI: https://doi.org/10.1145/3092703.3092727

[12] Al Danial. 2018. Count Lines of Code. Retrieved from https://github.com/AlDanial/cloc.

[13] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: Using names to refine types. See [39], 107–117. DOI: https://doi.org/10.1145/3236024.3236042

[14] 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, 48:1–48:4. DOI: https://doi.org/10.1145/2652524.2652567

[15] Daniela A. S. de Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr (Eds.). ACM, 296–305. DOI: https://doi.org/10.1145/2664243.2664254

[16] Sundar Dorai-Raj. 2014. *binom: Binomial Confidence Intervals for Several Parameterizations*. Retrieved from https://CRAN.R-project.org/package=binom.

[17] Facebook. 2014. Hack: A new programming language for HHVM. Retrieved from https://engineering.fb.com/developer-tools/hack-a-new-programming-language-for-hhvm.

[18] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Proceedings (FME'01)*. 500–517. DOI: https://doi.org/10.1007/3-540-45251-6_29

[19] 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'17)*. 758–769. DOI: https://doi.org/10.1109/ICSE.2017.75

[20] Marko Gasparic, Gail C. Murphy, and Francesco Ricci. 2017. A context model for IDE-based recommendation systems. *Journal of Systems and Software* 128 (2017), 200–219. DOI: https://doi.org/10.1016/j.jss.2016.09.012

[21] Narain Gehani. 1977. Units of measure as a data attribute. *Computer Languages* 2, 3 (1977), 93–111.

[22] Adam Gundry. 2015. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell'15)*. ACM, New York, NY, 11–22. DOI:https://doi.org/10.1145/2804302.2804305

[23] Philip Guo and Stephen McCamant. 2005. Annotation-less unit type inference for C. *Final Project, 6.883: Program Analysis* (2005).

[24] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Eric 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. DOI:https://doi.org/10.1007/s10664-013-9289-1

[25] Sudheendra Hangal and Monica S. Lam. 2009. Automatic dimension inference and checking for object-oriented programs. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. 155–165. DOI:https://doi.org/10.1109/ICSE.2009.5070517

[26] Cecil Hastings, Frederick Mosteller, John W. Tukey, Charles P. Winsor, et al. 1947. Low moments for small samples: A comparative study of order statistics. *Annals of Mathematical Statistics* 18, 3 (1947), 413–426.

[27] 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.

[28] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. See [39], 152–162. DOI:https://doi.org/10.1145/3236024.3236051

[29] Ronald T. House. 1983. A proposal for an extended form of type checking of expressions. *Computer Journal* 26, 4 (1983), 366–374.

[30] 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

[31] 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'06)*. 262–271. DOI:https://doi.org/10.1145/1134323

[32] 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. DOI:https://doi.org/10.1016/j.chb.2018.02.005

[33] Michael Karr and David B. Loveman III. 1978. Incorporation of units into programming languages. *Communications of the ACM* 21, 5 (May 1978), 385–391. DOI:https://doi.org/10.1145/359488.359501

[34] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian G. Elbaum, and Zhaogui Xu. 2018. Phys: Probabilistic physical unit assignment and inconsistency detection. See [39], 563–573. DOI:https://doi.org/10.1145/3236024.3236035

[35] Andrew Kennedy. 2009. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School - Third Summer School (CEFP'09), Revised Selected Lectures*. 268–305. DOI:https://doi.org/10.1007/978-3-642-17685-2_8

[36] Roger E. Kirk. 1982. *Experimental Design*. Wiley Online Library.

[37] 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'12)*. 153–162. DOI:https://doi.org/10.1109/ICPC.2012.6240483

[38] Makrina Viola Kosti, Kostas Georgiadis, Dimitrios A. Adamos, Nikos Laskaris, Diomidis Spinellis, and Lefteris Angelis. 2018. Towards an affordable brain computer interface for the assessment of programmers' mental workload. *International Journal of Human-Computer Studies* 115 (2018), 52–66. DOI:https://doi.org/10.1016/j.ijhcs.2018.03.002

[39] Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). 2018. *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'18)*. ACM. http://dl.acm.org/citation.cfm?id=3236024

[40] Yang Luo, Wanwangying Ma, Yanhui Li, Zhifei Chen, and Lin Chen. 2018. Recognizing potential runtime types from python docstrings. In *Software Analysis, Testing, and Evolution - 8th International Conference, Proceedings (SATE'18) (Lecture Notes in Computer Science)*, Lei Bu and Yingfei Xiong (Eds.), Vol. 11293. Springer, 68–84. DOI:https://doi.org/10.1007/978-3-030-04272-1_5

[41] R. Männer. 1986. Strong typing and physical units. *SIGPLAN Notices* 21, 3 (March 1986), 11–20. DOI:https://doi.org/10.1145/382280.382281

[42] 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. DOI:https://doi.org/10.1016/j.jss.2016.09.015

[43] Winter Mason and Siddharth Suri. 2012. Conducting behavioral research on Amazon's mechanical turk. *Behavior Research Methods* 44, 1 (2012), 1–23.

[44] Winter A. Mason and Duncan J. Watts. 2009. Financial incentives and the "performance of crowds." *SIGKDD Explorations* 11, 2 (2009), 100–108. DOI:https://doi.org/10.1145/1809400.1809422

[45] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Eric 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'12)*. 683–702. DOI:https://doi.org/10.1145/2384616.2384666

[46] McKee J. McClendon. 1991. Acquiescence and recency response-order effects in interview surveys. *Sociological Methods & Research* 20, 1 (1991), 60–103.

[47] Paul Mihas. 2019. Qualitative data analysis. In *Oxford Research Encyclopedia of Education.*

[48] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17, 3 (1978), 348–375. DOI:https://doi.org/10.1016/0022-0000(78)90014-4

[49] Jeremy W. Nimmer and Michael D. Ernst. 2002. Invariant inference for static checking: An empirical evaluation. *SIGSOFT Software Engineering Notes* 27, 6 (Nov. 2002), 11–20. DOI:https://doi.org/10.1145/605466.605469

[50] 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'17)*. ACM, New York, NY, 341–351. DOI:https://doi.org/10.1145/3092703.3092722

[51] 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'17)*. ACM, New York, NY, 352–355. DOI:https://doi.org/10.1145/3092703.3098219

[52] John-Paul 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'17)*. 712–718. DOI:https://doi.org/10.1109/IROS.2017.8202229

[53] John-Paul Ore, Sebastian G. Elbaum, Carrick Detweiler, and Lambros Karkazis. 2018. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. 190–201. DOI:https://doi.org/10.1145/3238147.3238173

[54] Sam Owre, Indranil Saha, and Natarajan Shankar. 2012. Automatic dimensional analysis of cyber-physical systems. In *Formal Methods (FM'12) (Lecture Notes in Computer Science)*, Dimitra Giannakopoulou and Dominique Méry (Eds.). Springer, Berlin, 356–371.

[55] 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'11)*. 199–209. DOI:https://doi.org/10.1145/2001420.2001445

[56] Grant W. Petty. 2001. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software: Practice and Experience* 31, 11 (2001), 1067–1076.

[57] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

[58] Lutz Prechelt and Walter F. Tichy. 1998. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering* 24, 4 (1998), 302–312. DOI:https://doi.org/10.1109/32.677186

[59] 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. 5.

[60] R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from http://www.R-project.org/.

[61] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 2015. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, Sriram K. Rajamani and David Walker (Eds.). ACM, 167–180. DOI:https://doi.org/10.1145/2676726.2676971

[62] Baishakhi Ray, Daryl Posnett, Premkumar T. Devanbu, and Vladimir Filkov. 2017. A large-scale study of programming languages and code quality in GitHub. *Commununications of the ACM* 60, 10 (2017), 91–100. DOI:https://doi.org/10.1145/3126905

[63] John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*. 408–423. DOI:https://doi.org/10.1007/3-540-06859-7_148

[64] G. Rosu and Feng Chen. 2003. Certifying measurement unit safety policy. In *18th IEEE International Conference on Automated Software Engineering, 2003, Proceedings*. 304–309. DOI:https://doi.org/10.1109/ASE.2003.1240326

[65] Pritam Roy and Natarajan Shankar. 2010. SimCheck: An expressive type system for Simulink. Retrieved from https://ntrs.nasa.gov/search.jsp?R=20100018536.

[66] Wasim Said, Jochen Quante, and Rainer Koschke. 2018. On state machine mining from embedded control software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*. IEEE Computer Society, 138–148. DOI:https://doi.org/10.1109/ICSME.2018.00024

[67] Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. 2012. Toward measuring program comprehension with functional magnetic resonance imaging. In *20th ACM SIGSOFT*

Symposium on the Foundations of Software Engineering (FSE-20) (SIGSOFT/FSE'12), Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 24. DOI: https://doi.org/10.1145/2393596.2393624

[68] Nicholas A. Smith, Isaac E. Sabat, Larry R. Martinez, Kayla Weaver, and Shi Xu. 2015. A convenient solution: Using MTurk to sample from hard-to-reach populations. *Industrial and Organizational Psychology* 8, 2 (2015), 220–228.

[69] 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)*. 99–108. DOI: https://doi.org/10.1145/2577080.2577098

[70] 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, 35:1–35:4. DOI: https://doi.org/10.1145/1852786.1852832

[71] 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'11)*. 97–106. DOI: https://doi.org/10.1145/2047849.2047861

[72] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, New York, NY, 456–468. DOI: https://doi.org/10.1145/2837614.2837630

[73] 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. DOI: https://doi.org/10.1016/j.chb.2017.08.038

[74] Paul A. Tipler and Gene Mosca. 2007. *Physics for Scientists and Engineers*. Macmillan.

[75] Amos Tversky and Daniel Kahneman. 1974. Judgment under uncertainty: Heuristics and biases. *Science* 185, 4157 (1974), 1124–1131.

[76] Zerksis D. Umrigar. 1994. Fully static dimensional analysis with C++. *SIGPLAN Notices* 29, 9 (1994), 135–139. DOI: https://doi.org/10.1145/185009.185036

[77] 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'15), Volume 1*. 234–245. DOI: https://doi.org/10.1109/ICSE.2015.44

[78] W. N. Venables and B. D. Ripley. 2002. *Modern Applied Statistics with S* (4th ed.). Springer, New York. Retrieved from http://www.stats.ox.ac.uk/pub/MASS4.

[79] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS'14), part of SLASH 2014*, Andrew P. Black and Laurence Tratt (Eds.). ACM, 45–56. DOI: https://doi.org/10.1145/2661088.2661101

[80] Mitchell Wand and Patrick O'Keefe. 1991. Automatic dimensional inference. In *Computational Logic - Essays in Honor of Alan Robinson*. 479–483.

[81] Jian Xiang, John C. Knight, and Kevin J. Sullivan. 2015. Real-world types and their application. In *Proceedings of the 34th International Conference on Computer Safety, Reliability, and Security - Volume 9337 (SAFECOMP'15)*. Springer-Verlag New York, Inc., New York, NY, 471–484.

[82] Jian Xiang, John C. Knight, and Kevin J. Sullivan. 2017. Is my software consistent with the real world? In *18th IEEE International Symposium on High Assurance Systems Engineering (HASE'17)*. IEEE Computer Society, 1–4. DOI: https://doi.org/10.1109/HASE.2017.20

Received May 2019; revised November 2020; accepted November 2020