The Behavior of Gradual Types: A User Study

Preston Tunnell Wilson
Brown University
Providence, Rhode Island
ptwilson@brown.edu

Ben Greenman
Northeastern University
Boston, Massachusetts
benjaminlgreenman@gmail.com

Justin Pombrio
Brown University
Providence, Rhode Island
jpombrio@cs.brown.edu

Shriram Krishnamurthi
Brown University
Providence, Rhode Island
sk@cs.brown.edu

Abstract

There are several different gradual typing semantics, reflecting different trade-offs between performance and type soundness guarantees. Notably absent, however, are any data on which of these semantics developers actually prefer.

We begin to rectify this shortcoming by surveying professional developers, computer science students, and Mechanical Turk workers on their preferences between three gradual typing semantics. These semantics reflect important points in the design space, corresponding to the behaviors of Typed Racket, TypeScript, and Reticulated Python. Our most important finding is that our respondents prefer a runtime semantics that fully enforces statically declared types.

ACM Reference Format:
Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: A User Study. In *Proceedings of Dynamic Languages Symposium (DLS'18)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In recent years, the long-standing debate between static and dynamic typing has been finding a reconciliation: gradual typing [27, 31]. In a gradually typed language, programmers are free to mix typed and untyped code. Some of the early gradually typed languages were created by retrofitting a type system on a (sublanguage of a) dynamic language (e.g., Typed Racket [32, 33], TypeScript [3], and Reticulated Python [36]); more recently, new languages are being made gradually typed from the outset, such as Pyret (`pyret.org`) and Dart 1 (`v1-dartlang-org.firebaseapp.com`).

But what should the semantics of a gradually-typed program be? In particular, when typed and untyped regions of code interact, what sort of runtime checks should protect the invariants of typed code? The answer to this question has implications for soundness, simplicity, performance, and (for retrofitted type systems) backward compatibility.

Several papers presenting these systems justify their designs by appealing to what they consider natural or intuitive to programmers [8, 9, 28, 33, 37]. However, none of the papers provide evidence to justify those claims. Our work repairs this weakness by performing the first study of developer preferences between different gradual typing semantics.

Concretely, we focus on three semantics: those corresponding to Typed Racket (Deep), TypeScript (Erasure), and Reticulated Python (Shallow). We adapt these semantics to a common surface language in the manner suggested by Greenman and Felleisen [12] and thus obtain three possibly-distinct behaviors for one mixed-typed program. Deep treats type annotations as (higher-order) contracts between regions of code. Erasure ignores type annotations at runtime. Shallow lies between these extremes; it checks values against type constructors.

We design a survey to illustrate key differences between the three semantics using short programs. We administer the survey to three populations: developers at a major software company, computer science students, and Mechanical Turk workers. We find that respondents generally dislike Erasure and like behavior that aligns with a statically-typed language. In addition to our findings, the survey is itself useful as a collection of representative programs that a language definition can use in its manuals to explain its runtime behavior.

2 Three Approaches to Gradual Typing

Soundness is a desirable property for any type system because it relates the ahead-of-time claims of the types to runtime outcomes. For example, if a sound type system claims that an expression e is of type $\mathsf{Int}\times\mathsf{Int}$ (representing a tuple of integers) and the evaluation of e yields a value, then the value is definitely a tuple with integer components. This fact about the tuple e can be used to state similar guarantees

DLS'18, November 2018, Boston, Massachusetts USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

about code that interacts with the tuple, and in general a programmer can use type soundness to reason compositionally about the correctness of a program.

A gradual typing system, however, cannot be sound in the normal sense because such systems let untyped values interact with statically-typed regions of code. For example, a typed module that imports an untyped value must declare a static type for the value, but cannot know until runtime whether the value matches the type. To illustrate, the typed code below expects a tuple of numbers but receives a tuple of strings at runtime:

```
 1 // UNTYPED code
 2 var f = function(x) { return (x, x); }
 3
 4 // TYPED code
 5 declare function
 6   f(x: String) : (Number, Number);
 7
 8 var nums : (Number, Number) =
 9   f("NaN");
10 var num  : Number =
11   nums[0];
```

The question for gradual typing is: how to defend a statically-typed context against a mismatched untyped value?

Three strategies have emerged: Deep, Erasure, and Shallow. In terms of the typed code above, which is internally type-correct, Erasure runs the program to completion despite the mismatch. Deep inserts an assertion that the call to `f` on line 9 returns a tuple of numbers and, because the tuple contains strings, halts before completing the assignment on line 8. Shallow only asserts that the call to `f` returns a tuple; because `f` does, this check passes. It later asserts that `nums[0]` on line 11 returns a number. The latter check fails and Shallow halts before the assignment on line 10. Generally, at runtime, Deep enforces types, Erasure ignores types, and Shallow enforces type constructors.

The following subsections outline the three strategies in more detail by explaining: (1) the motivation, (2) the source-code positions where runtime checks may occur, and (3) the nature of the runtime checks. Since each runtime check corresponds to a type, the examples assume a base type representing the set of integer values, an inductive type representing tuples, and a coinductive type for functions:

$$\tau \quad = \quad \mathsf{Int} \mid \tau{\times}\tau \mid \tau \rightarrow \tau$$

The reader may extrapolate an enforcement strategy for other base types (e.g., strings), inductive types (e.g., immutable sets), and coinductive types (e.g. arrays, objects).

2.1 Deep: Enforce Types

The goal of the Deep strategy is to offer a generalized notion of type soundness. Interactions between typed and untyped code may lead to a mismatch at runtime, but otherwise the programmer can trust the static types.

To this end, Deep strictly enforces the source-code boundaries between statically-typed and dynamically-typed code. If a typed context imports an untyped value, the value goes through a structural check. Dually, if an untyped context imports a typed function, the function receives latent protection against untyped inputs in the form of a derived type boundary. Runtime checks occur only at source-code boundaries and at derived boundaries for higher-order values.

When an untyped value v flows into a context that expects some value of type τ, written $v \Longrightarrow (\cdot : \tau)$, Deep employs the following type-directed validation strategy:

$\boxed{\text{Deep Strategy}}$

- $v \Longrightarrow (\cdot : \mathsf{Int})$
  check that v is an integer
- $v \Longrightarrow (\cdot : \tau_0{\times}\tau_1)$
  check that v is a tuple and recursively check its components; in particular, check that $v = \langle v_0, v_1\rangle$ and recursively check $v_i \Longrightarrow (\cdot : \tau_i)$ for each element
- $v \Longrightarrow (\cdot : \tau_d \rightarrow \tau_c)$
  check that v is a function and wrap v in a proxy that protects future inputs and checks future outputs (see Matthews and Findler [18] §3 for a discussion).

In summary, the Deep strategy *eagerly* checks finite values and *lazily* checks infinite values.

2.2 Erasure: Ignore Types

The Erasure strategy uses types for static analysis, and nothing more. At runtime, any value may flow into any context regardless of the type annotations:

$\boxed{\text{Erasure Strategy}}$

- $v \Longrightarrow (\cdot : \mathsf{Int})$
  check nothing
- $v \Longrightarrow (\cdot : \tau_0{\times}\tau_1)$
  check nothing
- $v \Longrightarrow (\cdot : \tau_d \rightarrow \tau_c)$
  check nothing

Despite the complete lack of type soundness, the Erasure strategy is popular among implementations of gradual typing. For one, the static type checker can point out logical errors in type-annotated code. Second, an IDE may use the static types in auto-completion and refactoring tools. Third, Erasure is simpler to implement than any form of type enforcement. Fourth, users that are familiar with the host language do not need to learn a new semantics to understand the behavior of type-annotated programs. Fifth, Erasure runs as fast as the original language.

2.3 Shallow: Protect Typed Code

The Shallow strategy ensures that typed code does not "go wrong" [22] in the sense of applying a primitive operation to a value outside its domain. For example, Shallow ensures that every function call targets a callable value.

In general, a "wrong" expression contains a value with an incorrect top-level shape. To prevent such expressions, it therefore suffices to check the top-level shape of values in three situations: (1) at the source-code boundaries between typed and untyped code, (2) before untyped code applies a typed function, and (3) after typed code receives a value from an untyped data structure or function. The SHALLOW strategy meets these requirements by defending statically-typed code. In particular, the defense adds one argument-check to the body of every typed function and guards every tuple projection and function application with a result check. The actual shape checks are simple:

$\boxed{\text{SHALLOW Strategy}}$

- $v \Longrightarrow (\cdot : \mathsf{Int})$
  check that v is an integer
- $v \Longrightarrow (\cdot : \tau_0 \times \tau_1)$
  check that v is a tuple
- $v \Longrightarrow (\cdot : \tau_d \rightarrow \tau_c)$
  check that v is a function

Informally, the SHALLOW strategy is a compromise between the hands-off attitude of ERASURE and the meticulous DEEP strategy. The SHALLOW type soundness guarantee, however, is weak and non-compositional. If a typed expression reduces to a value, the only certainty is that the value has the correct top-level shape.

3 Survey Method

We created a survey (the essence of which is in appendix A) consisting of several well-typed programs followed by the the program's behavior under each semantics. The purpose of the survey was to collect data on participants' preference between the DEEP, ERASURE, and SHALLOW behaviors.

We evaluated each behavior along two dimensions simultaneously: how subjects *felt* about the behavior (whether they Liked or Disliked it) and whether it matched their *expectations* (Expected or Unexpected). We call the combination of these (e.g., Like and Expected) an *attitude*. We presented the resulting four attitudes as the options for subjects to indicate their feeling about each behavior.

Observe that the dimensions are roughly independent. One might Like a particular behavior (say bignum arithmetic) but, since it is rarely seen in languages, find it Unexpected. One might even become habituated to behaviors they Dislike. For instance, a programmer might Dislike that + is not commutative in JavaScript but, having gotten accustomed to the behavior, may come to Expect it in other languages too (i.e., Dislike and Expected).

3.1 Survey Question Design

Designing an effective survey requires balancing several factors. Using large, existing programs has benefits, but: (a) the subjects' familiarity with and feelings towards the language could significantly affect our results; (b) non-semantic criteria like programming environments and error message presentation [2, 38] could be a major confounding factor—participants might evaluate these features instead of the different behaviors, as we discuss in section 6; and (c) our demands on subjects' time could be very high, resulting in little to no participation.

Instead, we created a multiple-choice quiz based on the possible interactions between typed and untyped regions of code. For the three kinds of types (base types, inductive types, and coinductive types) and two kinds of boundaries (typed-to-untyped and untyped-to-typed) this led to six basic boundary-crossing questions. After crossing one boundary, there are six second-order questions regarding the interactions between a context (typed or untyped) with a value (via reads from values of inductive type, and via reads and writes for values of coinductive type). Finally, we ask whether a value that crosses multiple type boundaries must live up all the types for the rest of the program, or only some.

From this exhaustive list of questions, we created eight small (3–6 line) programs from which we could infer an exhaustive set of answers. The programs were written in a conventional syntax. Our goal was to keep the number of questions small to minimize fatigue and loss of subject interest. Each program exhibits different behavior under at least two semantics, and the set as a whole tells all three apart (what Pombrio, et al. [25] call a "classifier"). We took the error messages from the corresponding representative DEEP and SHALLOW languages and distilled them down to a uniform format with a consistent amount of information (e.g., dropping the blame labels [11] provided in Typed Racket). We call these *error outputs* as opposed to error messages.

As part of the minimization effort, the survey uses syntax for only four kinds of values: integers, strings, arrays, and objects. The first two are values of base type, the latter two are values of coinductive type, and none of the above are naturally described by an inductive type. To collect attitudes for the different inductive-type checking strategies, the survey includes two questions in which one behavior checks the contents of array and objects that cross a boundary. Section 4 refers to this as a DEEP$*$ behavior.

In short, our programs were chosen to: (1) sample the gradual typing design space, (2) distinguish between behaviors, and (3) fit in a short survey. In section 6 we discuss several threats to validity and generalizability as a result of our approach, as well as some mitigating factors.

Additionally, the survey asks about: preference between typed and untyped programming, which typed languages participants had used, what languages they are comfortable with, what languages they use at work, how long they had been programming, what they find types useful for, whether they had ever used a gradually typed language, and whether they agreed with the statement "Type annotations should not change the behavior of a program" (to check whether participants agreed with an assumption of Erasure: see section 5.1). For the student population, we removed the question about work and instead asked which computer science courses they had taken at their university.

3.2 Survey Distribution

We administered this survey to three populations:

- Employees at a major Silicon Valley technology company (henceforth, "software engineers", or "S.E."), recruited by a former student now working there. We estimated a completion time of 20 minutes (based on student responses, below) and suggested advertising it as a "survey on programming language type system design". Since recruitment was done on an internal email list, we are not privy to further details. In four days (May 30–June 2), we received 34 responses.
- Computer science students at a highly selective, private US university ("students"). The survey was advertised on within-university social media and was kept open for two weeks (April 25–May 9). The first 25 students were offered a $10 Amazon gift card. We received 17 completions, not meeting our hoped-for 25 perhaps because the survey was only completed and tested around the time of final exams. The average completion time was 20 minutes, but subjects tended to cluster around 10, 20, and 30 minutes instead of being distributed uniformly.
- Workers on Mechanical Turk ("workers" or "Turkers"). The task ("HIT") was labeled "Answer a survey about types in a programming language—PRIOR PROGRAMMING EXPERIENCE REQUIRED". The survey was open for a week (June 13–June 20) and paid $2.50. The description mentioned that this survey was on a new programming language, and reiterated that prior programming experience was required. Internally, we thought that Turkers would spend five minutes on the survey, but in our description we gave the highest student average as an upper estimate on the time to complete (30 minutes). We recorded 186 responses. To eliminate bots and inattentive workers, we included an attention check midway through the survey. Besides the normal behaviors for a program, we added an answer saying "Attention check: select like and unexpected." After filtering those Turkers who failed this attention check, answered that they had never programmed before, or marked invalid gradually-typed or programming languages (such as "yes", "English", and "Spanish"), we had 90 remaining responses.

One of the software engineers had less than five years' experience; four had between 5–10 years' experience; and 29 had ten or more years' experience programming. The most common languages were Python (25), C++ (25), and Java (22), but subjects had experience with JavaScript (13), C# (9), Haskell (7), and other languages as well.

Four students had less than two years' experience; another four had between 2–5 years' experience; and nine had 5 or more years' experience. The median number of courses taken was nine. (We report the median because some students gave answers like "too many [courses] to count".) The most common languages were Java (16), Python (15), and C (7), with a smattering of other languages.

Fourteen of the Turkers had less than a year's experience, 26 had between 1–2 years' experience, 13 had between 2–5 years' experience, eight had between 5–10 years' experience, and 29 had ten or more years' experience. The most common languages were Java (40), Python (32), JavaScript (28), and C++ (27), out of around 40 languages in total.

4 Survey Results

Before you read further, we strongly encourage you to do the survey (Appendix A) yourself, so you can compare your answers to those of the subjects.

In this section, we present the results for each question individually.[1] We do so in two parts: the questions for which there is *consensus* (section 4.1), and the remaining questions, which are *contentious* (section 4.2). We define consensus for a question as a majority (>50%) of software engineers and a majority of students having the same attitude towards Deep and Erasure; the question is contentious otherwise.

Each question presents a small program and 2-3 behaviors for the program. We break down the responses for each question in the following order:

1. *By strategy:* Deep, Shallow, and Erasure. If a program has only two behaviors, then two of the strategies lead to the same behavior.
2. *By population:* S.E., Student, and MTurk.
3. *By attitude:* LE (Liked and Expected), LU (Liked and Unexpected), DE (Dislike and Expected), and DU (Dislike and Unexpected). The figures plot the percent of participants that selected each attitude.

4.1 Consensus

Looking at the first two questions (figs. 1a and 1b), we find:

- For both the software engineer and student populations, Erasure is both Disliked and Unexpected while the Deep (and the eager-checking Deep∗, introduced in section 3.1) behavior is Liked and Expected.
- The same consensus can still be seen (to a lesser extent) with the MTurk population.

For the second question, there is no majority attitude towards the Shallow behavior in any of the populations.

The software engineer and student populations Dislike Erasure and Shallow but Like Deep∗ for question 3 (fig. 1c).

[1] The full responses are available at: cs.brown.edu/research/plt/dl/dls2018/.

```
1 | var t = [4, 4];
2 | var x : Number = t;
3 | x
```

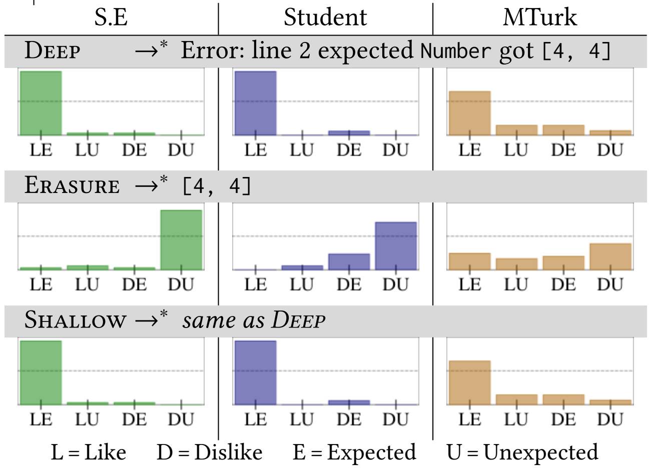


Figure 1a. Question 1 and responses

```
1 | var t = ["A", 3];
2 | var nums : Array(Number) = t;
3 | var fst1 : Number = nums[0];
4 | fst1
```

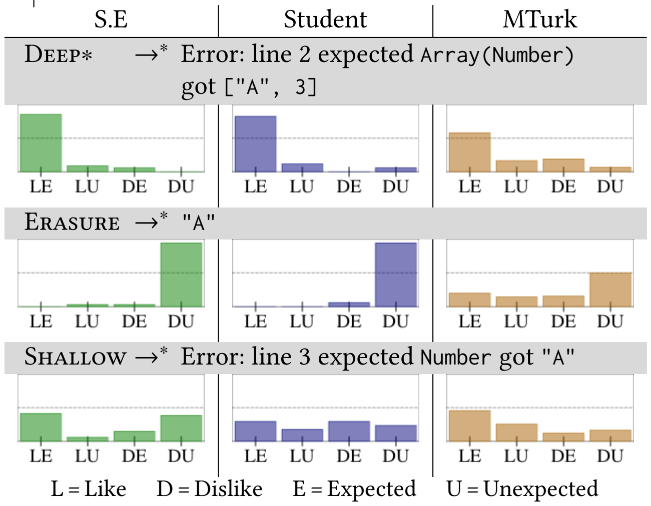


Figure 1b. Question 2 and responses

```
1 | var obj0 = {x = "A", y = 4};
2 | var obj1 : Object{x : Number, y : Number}
  |      = obj0;
3 | var y : Number = obj1.y;
4 | y
```

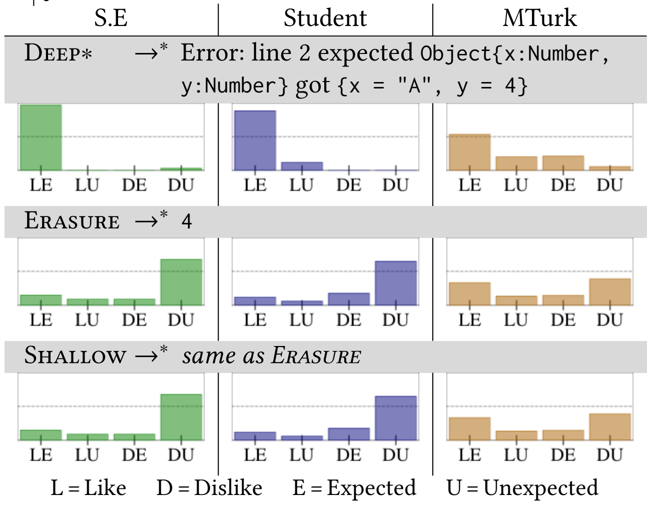


Figure 1c. Question 3 and responses

```
1 | var obj0 = {
  |     k = 0,
  |     add = function(i) { k = i } };
2 | var obj1 : Object{
  |     k : Number,
  |     add(i:String) : Void }
  |     = obj0;
3 | obj1.add("hello");
4 | var v : Number = obj1.k;
5 | v
```

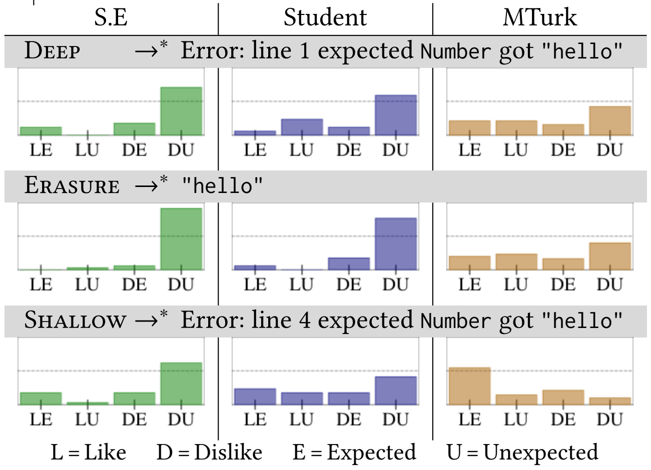
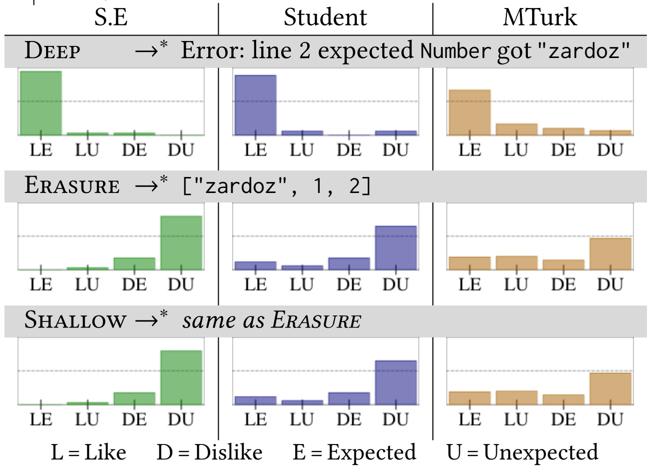


Figure 1d. Question 4 and responses

Question 4 (fig. 1d) is arguably our first complicated program. The main type mismatch is between a method of an object, obj0, and a typed object that is assigned to obj0. There are several places where the mistake can turn into an error since obj0 is untyped.

Interestingly, the way some of the respondents commented on DEEP's error output implied they did not trace the dynamic execution of the program. Instead, it appears as though they examined it statically. For example, some of them answered that "Line 1 does not reference hello in any way", seeming not to divine that add, declared on line 1, is executed after line 3 calls it. Many of our respondents included their reasoning for their particular selections: out of 34 software

```
1 | var nums : Array(Number) = [0, 1, 2];
2 | nums[0] = "zardoz";
3 | nums;
```



Deep $\to^*$ Error: line 2 expected `Number` got `"zardoz"`

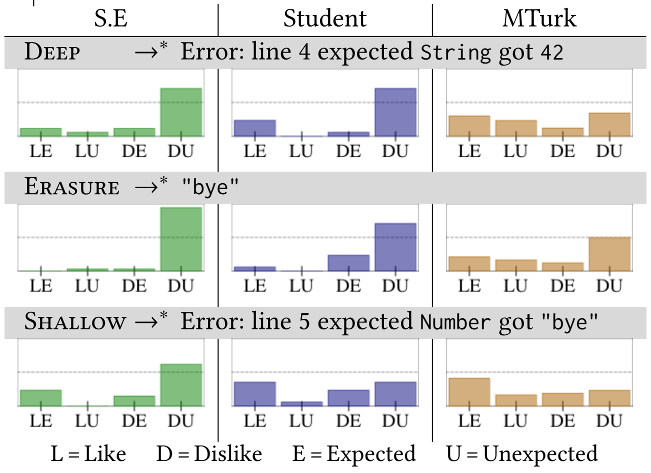
Erasure $\to^*$ `["zardoz", 1, 2]`

Shallow $\to^*$ *same as Erasure*

L = Like D = Dislike E = Expected U = Unexpected

Figure 1e. Question 6 and responses

```
1 | var x : Array(String) = ["hi", "bye"];
2 | var y = x;
3 | var z : Array(Number) = y;
4 | z[0] = 42;
5 | var a : Number = z[1];
6 | a
```



Deep $\to^*$ Error: line 4 expected `String` got `42`

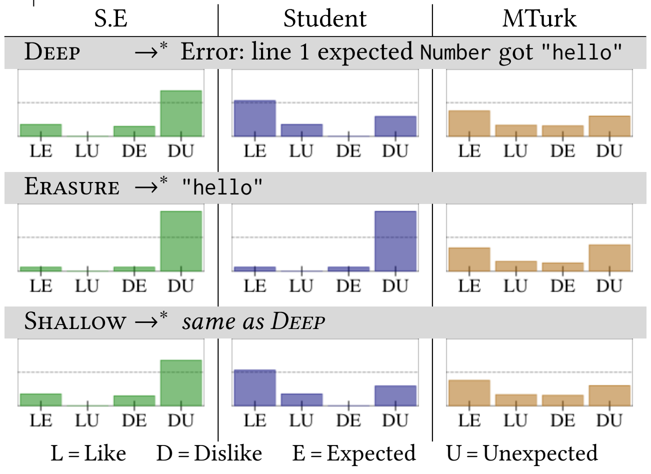
Erasure $\to^*$ `"bye"`

Shallow $\to^*$ Error: line 5 expected `Number` got `"bye"`

L = Like D = Dislike E = Expected U = Unexpected

Figure 1f. Question 7 and responses

```
1 | var obj0 = {
  |     k = 0,
  |     add = function(i : Number) { k = i }};
2 | var t = "hello";
3 | obj0.add(t);
4 | var k : String = obj0.k;
5 | k
```



Deep $\to^*$ Error: line 1 expected `Number` got `"hello"`

Erasure $\to^*$ `"hello"`

Shallow $\to^*$ *same as Deep*

L = Like D = Dislike E = Expected U = Unexpected

Figure 2a. Question 5 and responses

engineers, 24 left comments; out of 17 students, 11 left comments; out of 90 Turkers, 63 left comments. They reported that they expected the error on line 2 (15 for S.E.; 4 for students; 2 for Turkers) or on line 3 (7 for S.E.; 4 for students; 4 for Turkers). Line 2 corresponds to checking that add should not take in a `String` since it sets k to its parameter. Line 3 corresponds to the function application site. Several respondents noted that in a large program, it is essential to show which function application started the call in which the error occurs. We left out such a stack trace in the error output to keep our error outputs brief; we discuss this in section 6.

A majority of each population Likes the Deep behavior in question 6 (fig. 1e), and a majority of software engineers and students Dislike the Erasure and Shallow behavior.

Similar to question 4 (fig. 1d), question 7 (fig. 1f) is another case where a majority of both software engineers and students Disliked both behaviors. Twenty-four out of 34 of the software engineers Disliked all of the behaviors. Twenty-five of them Expected an error at line 3, where an array with an incorrect type annotation is assigned to an untyped array (as explained in their reasoning). Similarly, seven out of 17 students Expected an error at this location. Sixty out of 90 Turkers commented on this question. Seventeen of them Expected an error at line 3. We discuss the implications of so many participants Expecting an error at line 3 in more detail in section 7.

4.2 Contentious

In contrast to the "Consensus" questions, a majority of the software engineers and students have differing attitudes towards Deep and Erasure in the remaining questions.

For question 5 (fig. 2a), Deep's error output omits the function application site, similar to question 4 (fig. 1d). The 24 software engineers who explained their reasoning all expressed that the error should have been caught at line 3. Really, the error is due to a mismatch between the function definition on line 1 and the invocation on line 3; both line numbers should appear in a proper error message [38]. Of the

```
1 | var obj0 = {
  |     k = 0,
  |     add = function(i : Number) { k = i }};
2 | var t = "hello";
3 | obj0.add(t);
4 | var k : String = obj0.k;
5 | k
```

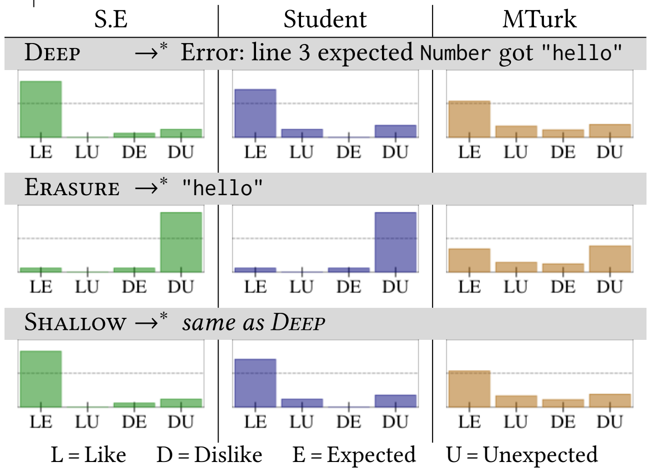


Figure 2b. Question 5' and responses

```
1 | var obj0 = {
  |     k = 0,
  |     update = function(i : Number) {k=i}};
2 | var obj1 = obj0;
3 | var obj2 : Object{
  |     k : Number,
  |     update(i : String) : Void }
  |     = obj1;
4 | obj2.update(4);
5 | var k : Number = obj2.k;
6 | k
```

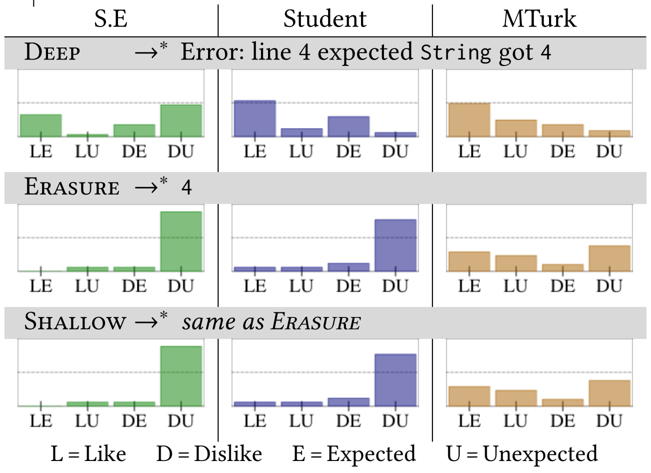


Figure 2c. Question 8 and responses

nine students who commented, five of them Expected an error at line 3. Fourteen out of the 65 Turkers who commented with their reasoning answered that the error should be at line 3. To us, "should" implies both Liking and Expecting this behavior. Because the behavior is the same, if we ignore the emphasis on line 1 in the error output, then these participants Like and Expect this behavior. Figure 2b presents this revised count and shows a preference for the Deep behavior.

Question 8 (fig. 2c) is similar to question 7 (fig. 1f)—we are testing to see what happens when one value crosses multiple boundaries between typed and untyped code. First, we create a typed object (`obj0`) and export the object to untyped code (`obj1`). We then import the object back into typed code (`obj2`), assigning an incompatible type to the object's `update` method. We then export the object to untyped code and call the `update` method with a value that matches the original type declaration. The Deep behavior detects a type mismatch; Erasure and Shallow forget the (incompatible) type for `obj2`. We see that a fair number of participants Expect the error to happen on line 3 due to the object assignment: out of the 23 software engineers who explained their reasoning, 19 of them Expected an error here; out of six students who commented, four of them Expected the error to be here. Out of the 57 Turkers who explained their reasoning, three Expected an error here.

5 Preference Analyses

Designers of languages might want to seek preference information from developers regarding certain design decisions. However, this information is potentially misleading if developers act differently on concrete programs compared to how they answer abstract questions. We compare developers' attitudes towards behaviors to their opinion on three background questions: whether type annotations should change program behavior, whether they prefer typed or untyped programming, and whether they have used a gradually-typed language before.

5.1 Type Annotations

We asked respondents if they agreed or disagreed with the statement: "Type annotations should not change the behavior of a program". Then for respondents that are *consistent* in their attitudes towards Erasure, we check to see if their attitude matches their answer to the background question. From this we can judge whether participants are *accurate* or *inaccurate* in self-reporting. We define being *consistent* for a behavior (e.g., Erasure) to mean Liking the behavior at least six times (out of eight) or Disliking the behavior at least six times.[2]

We tabulate the consistent subjects' opinion on Erasure and their opinion on the effect of type annotations, resulting

[2]We do not filter for consistency on Deep or Shallow because one might agree that types should affect behavior but disagree with the particular behaviors exhibited by Deep and Shallow.

		Type annotations should not change the behavior of a program.	
Pop.	Erasure Opinion	Agree	Disagree
S.E.	Like	1	0
	Dislike	14	19
Student	Like	0	0
	Dislike	2	14
MTurk	Like	19	11
	Dislike	18	25

Table 1. Annotation preference by Opinion on Erasure

		Which do you prefer: typed or untyped programming? (coded)			
Pop.	Erasure Opinion	Un-typed	Typed	Gradual	Size Dep.
S.E.	Like	1	0	0	0
	Dislike	0	26	1	6
Student	Like	0	0	0	0
	Dislike	1	15	0	0
MTurk	Like	6	24	0	0
	Dislike	8	32	0	3

Table 2. Type Preference by Opinion on Erasure

in table 1. Summing the main diagonal of each sub-table gives us the number of participants who are accurate in each population: 20 for S.E., 14 for Student, and 44 for MTurk. The anti-diagonal gives us the number of inaccurate participants: 14 for S.E., 2 for Student, and 29 for MTurk.

We see that most of the inaccurate participants fall into the lower-left part of their sub-table. These respondents Agree, in the abstract, that type annotations should not change the behavior of a program, but when faced with actual programs, Dislike Erasure.

5.2 Typed/Untyped Programming Preference

Is attitude towards a behavior tied to respondents' preferences on typed or untyped programming? For each population of consistent users, we look at their preference for typed versus untyped programming based on the background question. Since the question is open-ended, we code (in the social science sense) participants' responses for whether they prefer typed, untyped, or gradually typed programming, or if their preference is dependent on the size of the code. Tabulating the code for a participant and their opinion on Erasure results in table 2.

Most participants' opinion matches their programming preference, except for the 24 Turkers who Like Erasure behavior but prefer typed programming. Fourteen of these Turkers consistently Like Deep behavior too, so we suspect they either misunderstood the survey or are happy with all kinds of behaviors.

We are also interested in whether programming preference is related to having a specific attitude towards a behavior for a question. For each combination of population, question, behavior, and attitude, we count the number of users represented in table 2 who selected the attitude, then split this count according to their coded preference for typed or untyped programming. We use Fisher's Exact test to see if participants with different preferences have different attitudes. Since we are doing so many comparisons, we adjust our critical value (originally $\alpha = 0.05$) with a Bonferroni correction for the number of different behaviors per question. For example, if Deep and Shallow produce the same output, then there are only two different behaviors for that question, so our adjusted critical value would be $\alpha = 0.025$.

Based on the corrected Fisher test, we do not find any significant relationship between attitude towards a behavior on a question and programming preference for any population.

5.3 Gradual Typing Exposure

We analyze whether prior gradual typing exposure is correlated with Deep, Erasure, or Shallow being Expected or Unexpected. We perform this analysis per behavior per question since it is possible that experience has familiarized a participant with one behavior but not the others. We test using Fisher's Exact test and use the Bonferroni correction per question to adjust our critical value. We do not find any significant relationship between prior exposure to gradual typing and finding the behavior Expected or Unexpected for any population, over any question, over any behavior.

6 Threats to Validity

It is possible that participants answered our survey not in response to the questions and behaviors we tested but something else instead. We outline some of these concerns that might have affected the validity of the study.

We used a feint to elicit participants' attitudes towards Deep, Erasure, and Shallow; namely, the introduction to our survey claimed we were designing a new language. It is possible that respondents exaggerated in their comments or that they misrepresented their opinions to try to influence the presumed language.

Some of our analysis relied on whether subjects thought type annotations should change program behavior. It is possible, though, that we were measuring their definition of "type annotations" or "programs" rather than their attitudes.[3] Several of the software engineers commented that type annotations should not change the behavior of "correct" programs and should have compile-time errors for those that have type-incompatible operations. One software engineer wrote that

[3]For example, we could have tried this study with Javadoc-style annotations to see if their opinions change.

they would expect *typing* a program to potentially change its behavior but that adding type *annotations* should not, taking annotations as a weaker version of type declarations.

Another ambiguous phrase that might have affected our results is "type inference." There are several versions of type inference in languages, from unification in Haskell and ML-languages to local type inference in C++, Rust, and Go. Thus, saying our language does not have type inference could mean different things to different respondents.

The way that respondents read the programs and error outputs also could have affected our validity as we discussed in section 4.1. For programs that cited the declaration of a function where the program crashed rather than the function call site, several respondents mentioned that the declaration line did not mention the improperly typed value. It appears they did not see the behaviors as results of running the program but rather as static type errors.

Some participants might have been confused whether the errors were runtime or static errors; we tried to prevent this by saying that the programs passed the static type checker (appendix A).

Threats to Generalizability The software engineers and students who responded work for a highly selective software engineering company and attend a highly selective university, respectively. Other developers' and students' opinions could be very different. On the other side of the spectrum, we have the MTurk population, with a wide variety of experience and different incentives for taking the survey, so we avoid general statements about the Turkers.

Another issue is ecological validity: participants are answering questions instead of developing software, programs are 3–6 lines long instead of hundreds or thousands spread across different files, and programs produce error outputs instead of error messages with detailed stack traces and blame information. The interplay between types and refactoring is not reflected in these questions. Some of these aspects we give up to get more opinions on behaviors (see section 3 for rationale). Regarding size, if a participant dislikes a behavior when they can keep the program in their head, then they will probably still dislike it when the program spans thousands of lines and multiple files.

One might hypothesize that blame information would clarify program behavior, but then the error message would be a major confounding factor. If we include blame information, we might accidentally measure how participants like its presentation, which would be a different threat to validity. To try to measure only the behavior, we keep our error outputs minimal and consistent between behaviors.

We paid two populations to complete this survey, which might have changed their responses. All of the respondents were self-selecting, which might introduce some bias.

It is possible, but unlikely, that developers would not run into these types of programs. Our experience is that as code grows, the frequency of typed/untyped interaction increases and leads to the situations these programs represent.

Finally, the survey does not suggest any performance consequences for different behaviors, despite evidence that Shallow adds overhead relative to Erasure [13] and that Deep can slow a program by an order of magnitude [30]. We could not think of a way to ask such questions about our hypothetical language in a meaningful manner; thus our results reflect attitudes regarding semantics only, ignoring performance. Three respondents mentioned the cost of checking types at runtime (two were explaining their reasoning for a question; the other was commenting on the language in general). Other subjects could have implicitly factored performance into their answers, but we cannot tell from their responses.

7 Language Design Implications

Developers do not expect a runtime system to ignore type annotations—we see this in the majority attitude towards Erasure for all questions for both the software engineers and the students. The designers of Erasure systems must therefore clearly communicate this choice to potential users.

The responses to questions 2 and 7 (figs. 1b and 1f) suggest that developers prefer a strategy that type-checks the elements of an untyped array immediately when the array flows into a typed context (Deep∗) instead of waiting until the array is accessed (Deep).[4] Twenty-five out of 34 software engineers (74%) commented on these questions, saying that they Expected an error where the untyped array value was assigned a type. Seven out of 17 students (41%) also Expected an error at this location. Seventeen out of the 60 Turkers (28%) who commented wanted an error at this location.

The responses to questions 7 and 8 (fig. 1f, fig. 2c) suggest that a gradual typing system should remember the type associated with a value even when it flows to an untyped context. In particular, if a typed array flows to untyped code and then flows back in to typed code at a different type, the original type should also be enforced.

8 Related Work

8.1 User Studies

Miller et al. [21] found that there can be significant differences between a language as specified, as implemented, and as understood by its users. They improve the Yedalog language by making its specification more closely match its user's expectations. Our study is an initial step to finding what programmers expect from a gradual type system; an eventual goal is to have gradually-typed languages match users' expectations.

Tunnell Wilson et al. [34] have previously tried crowd-sourcing language design decisions using Mechanical Turk.

[4]Typed Racket implements Deep, but its blame system reports the error in terms of the point where the untyped value flowed in. This provides an error similar to Deep∗, but is delayed until an element is accessed.

They created surveys for a variety of language features to investigate Turkers' expectations and consistency. Their surveys did not, however, cover gradual typing. More importantly, their survey lacked other populations as a reference, whereas we survey two other populations. We see that Turkers are not good proxies for software engineers at an elite company. (If anything, students appear to be slightly better proxies. On one hand, this is unsurprising because many of these students go on to jobs at similar companies. On the other hand, they lack the experience of those developers.)

Several researchers have studied the effect of *static* type systems on development [10, 14, 15, 19], finding that they reduce development time, help in maintainability, and aid in using undocumented functions and APIs. These findings motivate static types in gradually typed development too. However, we are not measuring the effects of static types in gradually typed languages but instead are getting developers' attitudes towards different behaviors possible with static type declarations.

Mezzetti et al. [20] studied whether Dart 1's unsound type system[5] really does meet its design goals of being pragmatic and not getting in the way of its users. They describe soundness in Dart 1, show sound alternatives, and report the number of static type errors (called "warnings") their altered type-checker and runtime make. They conclude that some of these sound alternatives would catch actual programmer error while not introducing too many false-positives.

Groovy has optional type annotation checking. Souza and Figueiredo [29] study how programmers use optional typing in a repository of Groovy programs. Type annotation checking was a recent feature for Groovy at the time, so type annotations in the code served mostly as documentation. They found that programmers use type annotations more frequently in the interface of their modules, less frequently in test classes and scripts, and that programmers' use of types is inversely correlated to whether they have programmed in a dynamically typed language before. We do not ask developers how they use optional types, but we do try to relate participants' responses with their preferences and exposure to gradually typed languages.

The Natural Programming project [24] took the language children and adults use to solve problems as inspiration in designing their language. We focus on one feature of a language and survey people with programming experience, but the purpose of our studies is the same: seeing what participants think is natural (or in our case, Liked and Expected).

8.2 Gradual Typing

The DEEP approach originates in the observation that higher-order contracts [11] can dynamically enforce higher-order types [16, 18, 31]. Typed Racket [32, 33], Gradualtalk [1], and TPD [37] are three DEEP systems.

The ERASURE approach corresponds to *optional typing* [6]. An optional (or *pluggable* [6]) type checker uses type annotations only for static analysis; adding or removing annotations does not affect the behavior of a program. Examples include: Flow (`flow.org`), Pytype (`github.com/google/pytype`), rtc [26], Hack (`hacklang.org`), Pyre (`pyre-check.org`), Typed Clojure [5], mypy (`mypy-lang.org`), TypeScript [3], and Typed Lua [17].

The SHALLOW approach is based on the *transient* semantics of Reticulated Python [35, 36]. Transient enforces type constructors within typed code; it protects every typed context against possibly-untyped values using first-order checks. Pyret uses a similar strategy to enforce user-supplied type annotations. Every annotation in a Pyret program corresponds to a runtime type-constructor check.

A fourth approach to gradual typing is the *concrete* approach pioneered in Thorn [39], implemented at scale in C# [4] and Dart 2 (`dartlang.org/dart-2`), and adapted to satisfy the gradual guarantee [28] in Nom [23]. Unlike the three approaches considered in this paper, every value in a concrete language comes with a static type (Chung et al. [7] present a formal comparison). This invariant makes it possible to enforce types with first-order checks. To our knowledge, no study has asked programmers for their preference between a language implementing the concrete approach and a language with equally-expressive untyped code. However, some of the questions in this survey cannot be asked in the concrete approach, which does not support, e.g., passing an untyped object into typed code.

9 Conclusions

Asking developers for feedback on program behaviors is useful to refine and test design decisions. We have evidence that professional developers and students dislike behaviors that do not enforce statically declared types at runtime for small programs (and expect that they would not like it for larger programs spread across modules and files, either).

We also find that software engineers and students have different attitudes towards these gradual typing behaviors. Given the variety of opinions, it is essential that there is clear documentation so that programmers understand how their language behaves—especially if the behavior is not what they expect. A combination of both positive and negative examples would help with this. We propose our survey as a minimal set of programs demonstrating language behavior.

Acknowledgments

We would like to thank: all of our participants for putting their time and effort into answering our survey, Frank Goodman for distributing our survey, Leif Andersen for helping design our figures, and the anonymous DLS reviewers for their feedback. This work was partially supported by NSF grant CCF-1518844.

[5]Dart 2 is type safe: `dartlang.org/guides/language/sound-dart`

[6]See manuscript at: `bracha.org/pluggableTypesPosition.pdf`

A The Survey

The following is a simplified LaTeX reproduction of the surveys we released through Qualtrics. To view the surveys in HTML exactly as participants saw them, visit:

cs.brown.edu/research/plt/dl/dls2018/

Key: L = Like D = Dislike E = Expected U = Unexpected

We are designing a language that mixes typed and untyped code. We want your opinion on what should happen when untyped values flow into typed expressions.

Our language has static type checking but does not have type inference. For example, this program would pass the static type checker but would error at line 3:

```
1  var x : Number = 4;
2  var y = "hello";
3  x / y
```

The following 8 questions ask your opinion about possible results of running a few programs that pass the static type checker (but may still have runtime errors). We are not looking for feedback on syntax.

Question 1

```
1  var t = [4, 4];
2  var x : Number = t;
3  x
```

	LE	LU	DE	DU
Error: line 2 expected `Number` got `[4, 4]`	○	○	○	○
`[4, 4]`	○	○	○	○

Question 2

```
1  var t = ["A", 3];
2  var nums : Array(Number) = t;
3  var fst1 : Number = nums[0];
4  fst1
```

	LE	LU	DE	DU
Error: line 2 expected `Array(Number)` got `["A", 3]`	○	○	○	○
Error: line 3 expected `Number` got `"A"`	○	○	○	○
`"A"`	○	○	○	○

Question 3

```
1  var obj0 = {x = "A", y = 4};
2  var obj1 : Object{x : Number, y : Number}
       = obj0;
3  var y : Number = obj1.y;
4  y
```

	LE	LU	DE	DU
Error: line 2 expected `Object{x:Number, y:Number}` got `{x = "A", y = 4}`	○	○	○	○
`4`	○	○	○	○

Question 4

```
1  var obj0 = {
       k = 0,
       add = function(i) { k = i } };
2  var obj1 : Object{
       k : Number,
       add(i:String) : Void }
       = obj0;
3  obj1.add("hello");
4  var v : Number = obj1.k;
5  v
```

	LE	LU	DE	DU
Error: line 1 expected `Number` got `"hello"`	○	○	○	○
Error: line 4 expected `Number` got `"hello"`	○	○	○	○
`"hello"`	○	○	○	○
(Turk only) Attention check: select LU	○	○	○	○

Question 5

```
1  var obj0 = {
       k = 0,
       add = function(i : Number) { k = i }};
2  var t = "hello";
3  obj0.add(t);
4  var k : String = obj0.k;
5  k
```

	LE	LU	DE	DU
Error: line 1 expected `Number` got `"hello"`	○	○	○	○
`"hello"`	○	○	○	○

Question 6

```
1  var nums : Array(Number) = [0, 1, 2];
2  nums[0] = "zardoz";
3  nums;
```

	LE	LU	DE	DU
Error: line 2 expected `Number` got `"zardoz"`	○	○	○	○
`["zardoz", 1, 2]`	○	○	○	○

Question 7

```
1  var x : Array(String) = ["hi", "bye"];
2  var y = x;
3  var z : Array(Number) = y;
4  z[0] = 42;
5  var a : Number = z[1];
6  a
```

	LE	LU	DE	DU
Error: line 4 expected `String` got 42	○	○	○	○
Error: line 5 expected `Number` got `"bye"`	○	○	○	○
`"bye"`	○	○	○	○

Question 8

```
1  var obj0 = {
       k = 0,
       update = function(i : Number) {k=i}};
2  var obj1 = obj0;
3  var obj2 : Object{
       k : Number,
       update(i : String) : Void }
       = obj1;
```

```
4 │obj2.update(4);
5 │var k : Number = obj2.k;
6 │k
```

	LE	LU	DE	DU
Error: line 4 expected `String` got 4	○	○	○	○
4	○	○	○	○

References

[1] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2013. Gradual typing for Smalltalk. *Science of Computer Programming* 96, 1 (2013), 52–69.

[2] Titus Barik. 2018. *Error Messages as Rational Reconstructions.* Ph.D. Dissertation. North Carolina State University.

[3] Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*. 257–281.

[4] Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *ECOOP*. 76–100.

[5] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *ESOP*. 68–94.

[6] Gilad Bracha and David Griswold. 1993. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA*. 215–230.

[7] Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In *ECOOP*. 12:1–12:23.

[8] Matteo Cimini and Jeremy Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *POPL*. 789–803.

[9] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *ESOP*. 214–233.

[10] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How Do API Documentation and Static Typing Affect API Usability?. In *ICSE*. 632–642.

[11] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP*. 48–59.

[12] Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. 2, ICFP (2018), 71:1–71:32.

[13] Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-Tag Soundness. In *PEPM*. 30–39.

[14] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. 19, 5 (2014), 1335–1382.

[15] 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 *ICPC*. 153–162.

[16] Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *TOPLAS* 32, 6 (2010), 1–34.

[17] Andre Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2015. A Formalization of Typed Lua. In *DLS*. 13–25.

[18] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *TOPLAS* 31, 3 (2009), 1–44.

[19] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In *OOPSLA*. 683–702.

[20] Gianluca Mezzetti, Anders Møller, and Fabio Strocco. 2016. Type Unsoundness in Practice: An Empirical Study of Dart. In *DLS*. 13–24.

[21] Mark S Miller, Daniel Von Dincklage, Vuk Ercegovac, and Brian Chin. 2017. Uncanny Valleys in Declarative Language Design. In *SNAPL*. 9:1–9:12.

[22] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences*, Vol. 17. 348–375.

[23] Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. In *OOPSLA*. 56:1–56:30.

[24] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environment. *Commun. ACM* (2004).

[25] Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler. 2017. Teaching Programming Languages by Experimental and Adversarial Thinking. In *SNAPL*. 13:1–13:9.

[26] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The Ruby Type Checker. In *SAC*. 1565–1572.

[27] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming*. 81–92.

[28] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL*. 274–293.

[29] Carlos Souza and Eduardo Figueiredo. 2014. How Do Programmers Use Optional Typing?: An Empirical Study. In *International Conference on Modularity*. 109–120.

[30] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *POPL*. 456–468.

[31] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *DLS*. 964–974.

[32] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL*. 395–406.

[33] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten years later. In *SNAPL*. 17:1–17:17.

[34] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *SPLASH Onward!* 1–17.

[35] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *DLS*. 45–56.

[36] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *POPL*. 762–774.

[37] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *ECOOP*. 28:1–28:29.

[38] John Wrenn and Shriram Krishnamurthi. 2017. Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages. In *Onward!* 134–147.

[39] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Ostlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL*. 377–388.