On the Cost of Type-Tag Soundness

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

Zeina Migeed
Northeastern University
Boston, Massachusetts, USA
migeed.z@outlook.com

Abstract

Gradual typing systems ensure type soundness by transforming static type annotations into run-time checks. These checks provide semantic guarantees, but may come at a large cost in performance. In particular, recent work by Takikawa et al. suggests that enforcing a conventional form of type soundness may slow a program by two orders of magnitude.

Since different gradual typing systems satisfy different notions of soundness, the question then arises: what is the cost of such varying notions of soundness? This paper answers an instance of this question by applying Takikawa et al.'s evaluation method to Reticulated Python, which satisfies a notion of type-tag soundness. We find that the cost of soundness in Reticulated is at most one order of magnitude, and increases linearly with the number of type annotations.

CCS Concepts • **General and reference** → **Metrics**; • **Software and its engineering** → *Software evolution*;

Keywords Migratory typing, Performance evaluation, Tag soundness, D-deliverable, Type granularity

ACM Reference Format:
Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-Tag Soundness. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3162066

1 How Much does Soundness Cost?

Gradual typing systems can help programmers with the task of maintaining code written in a dynamically typed language. If the language comes with a gradual typing system, developers may incrementally add type annotations as they improve a piece of the code base. The next developer that needs to comprehend this part of the application can use the type annotations to understand its structure and invariants.

While gradual typing can improve readability and robustness, it has serious implications for performance. The problem is that gradual typing systems enforce type soundness with run-time assertions that check whether values supplied by dynamically typed code match the type system's assumptions. These checks can impose a large performance cost.

Since the design space of gradual typing comes with a range of soundness notions, the question arises how much soundness costs in terms of performance. One such notion is Typed Racket's generalized type soundness [6]. At a high level, generalized soundness states that if a well-typed term reduces to a value, the value has the expected type. Otherwise, evaluation halts with a type error that directs the programmer to the source of the unexpected value. The performance cost of this guarantee is evidently high. An evaluation by Takikawa et al. [4] found that Typed Racket's implementation of generalized soundness can slow a working program by over two orders of magnitude.

A second notion of gradual type soundness is Reticulated's tag soundness [8]. Tag soundness guarantees that if a well-typed expression reduces to a value, then the value has the correct top-level type constructor (see section 2). Thus an expression with type `List(Int)` may reduce to a list of strings, but not to an integer or a function.

One might expect that gradual typing in Reticulated comes at a lower performance cost, but this claim has not been systematically evaluated. For example, both Vitousek et al. [8] and Muehlboeck and Tate [2] report the performance of Reticulated on fully-typed and fully-untyped programs, but do not report the performance of programs that actually use gradual typing. Part of the challenge is that Reticulated supports the addition of type annotations at a fine granularity, making exhastive evaluation infeasible for many programs. We address this limitation with an evaluation method based on random sampling (see section 3.1 and the appendix).

This paper contributes a systematic evaluation of the cost of gradual typing in Reticulated. The central findings are:

- Reticulated experiences a slow down of at most one order of magnitude at a function-level granularity;
- the performance degradation is approximately a linear function of the number of type annotations; and
- random sampling can approximate the performance overhead of gradual typing in Reticulated with a linear number of samples from an exponentially-large space.

PEPM'18, January 8–9, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5587-2/18/01…$15.00
https://doi.org/10.1145/3162066

```
@fields({"dollars": Int, "cents":Int})
class Cash:
  def __init__(self:Cash, d:Int, c:Int)->Void:
    self.dollars = d
    self.cents = c

  def add_cash(self:Cash, other:Cash)->Void:
    self.dollars += other.dollars
    self.cents += other.cents
```

Figure 1: A well-typed class

Outline: Section 2 introduces Reticulated, Section 3 adapts the Takikawa method, and Section 4 presents the evaluation.

2 Reticulated Python

Reticulated is a gradual typing system for Python[1] that gives programmers the ability to annotate functions and class fields with types [7, 8]. By way of example, figure 1 presents a type-annotated class representing US currency. The annotations imply two high-level invariants: (1) instances of the `Cash` class have integer-valued fields, and (2) the `add_cash` method is only invoked with instances of the `Cash` class.

Within the `add_cash` method, Reticulated enforces these invariants by translating the type annotations into dynamic checks that protect the two arguments of `add_cash` and the four dereferences of the fields `dollars` and `cents` [7]. These checks defend the statically typed method from arbitrary callers. If a Python context invokes `add_cash` with an integer, then the program will halt with a *type-tag error*.

2.1 Tag Soundness

Reticulated uses dynamic type checks to implement a form of type soundness [8]. Informally, if `e` is a well-typed expression, then evaluating `e` can result in one of four outcomes:

1. the program execution terminates with a value `v` that has the same type tag as the expression `e`;
2. the execution diverges;
3. the execution ends in an exception due to a partial computational primitive (e.g., division-by-zero); or
4. the execution ends in a type-tag error.

A *type tag* is essentially a type constructor without parameters. For completeness, figure 2 presents selected types τ and tags κ, as well as the mapping $\lfloor\cdot\rfloor$ from types to tags.[2]

Tag soundness is clearly weaker than standard type soundness; a well-typed program can reduce to a value that does not match its static type annotation. Figure 3 demonstrates with an expression that has the static type `List(Int)` but evaluates to a list containing a string and a function. This

[1] Specifically, CPython 3.

[2] The type Dyn is the dynamic type. Every expression is well-typed at Dyn.

$$\begin{array}{lcl} \tau & = & \mathsf{Int} \mid \mathsf{List}(\tau) \mid \mathsf{Function}([\tau], \tau) \mid \mathsf{Dyn} \\ \kappa & = & \mathsf{Int} \mid \mathsf{List} \mid \mathsf{Function} \mid \mathsf{Dyn} \end{array}$$

$$\boxed{\lfloor\tau\rfloor = \kappa}$$

$$\lfloor\mathsf{Int}\rfloor = \mathsf{Int} \qquad \lfloor\mathsf{Function}([\tau], \tau')\rfloor = \mathsf{Function}$$
$$\lfloor\mathsf{List}(\tau)\rfloor = \mathsf{List} \qquad \lfloor\mathsf{Dyn}\rfloor = \mathsf{Dyn}$$

Figure 2: Selected types (τ) and type tags (κ)

```
def make_ints()->List(Int):
  xs = []
  xs.append("NaN")
  xs.append(make_ints)
  return xs

make_ints() # returns ["NaN", <function>]
```

Figure 3: A strange but well-typed function

particular program succeeds because the `append` method is dynamically typed, but the general issue is that Reticulated supports only tag-level compositional reasoning. A programmer cannot trust the types beyond their top-level constructor.

Nevertheless, tag soundness is a pragmatic guarantee to retrofit to a dynamically-typed language. Reticulated's main design goal is to provide seamless interaction with the Python 3 runtime and libraries [3]. Consequently, Reticulated cannot implement a standard form of type soundness. There are two fundamental reasons why Reticulated must aim for a different guarantee.

First, any interaction between Reticulated code and Python code can potentially cause a type-tag error. There are two reasons for this. On one hand, the Reticulated type annotation might not match the behaviors implemented by the Python code. On the other hand, the Python code might contain a bug. These impedance mismatches cannot be caught without analyzing the Python code, and so the fourth clause of tag soundness admits the possibility of tag errors.

Second, Python code may inspect the representation of values. Reticulated must therefore ensure that a value from statically-typed code is indistinguishable from a Python value. The only way to meet this criterion (without modifying the Python runtime API) is to use the same value in both cases.[3] In particular, a Reticulated list must be indistinguishable from a Python list. This indistinguishability constraint explains why it is difficult for Reticulated to predict the run-time type of a value.

Reticulated chooses to implement tag soundness instead of some other compromise because of a secondary design

[3] Other gradually-typed languages use proxies to approximate indistinguishability [5, 9]. This approach typically fails when values are serialized or sent across a foreign function interface (FFI).

goal: all dynamic type checks run in near-constant time.[4] Instead of checking the type of values within a data structure, Reticulated stops at the structure's outermost tag. Hence list types require an $\Theta(1)$ tag check and structural object types with f fields require an $\Theta(f)$ check that the given value binds the proper fields. Intuitively, such checks should impose little overhead no matter how a programmer adds type annotations.

3 Evaluation Method

Takikawa et al. [4] introduce a three-step method for evaluating the performance of a gradual typing system: (1) identify a suite of fully-typed programs; (2) measure the performance of all gradually-typed *configurations* of the programs; (3) count the number of configurations with performance overhead no greater than a certain limit. They apply this method to Typed Racket, a gradual typing system with module-level granularity; in other words, a Typed Racket program with M modules has 2^M gradually-typed configurations.

Reticulated supports gradual typing at a much finer granularity, making it impractical to directly apply the Takikawa method. A naive application would require 2^a measurements for one function with a formal parameters, and similarly 2^f measurements for one class with f fields. The following subsections therefore generalize the Takikawa method (section 3.1) and describe the protocol we use to evaluate Reticulated (section 3.2).

3.1 Generalizing the Takikawa Method

A gradual typing system enriches a dynamically typed language with a notion of static typing; that is, some pieces of a program can be statically typed. The *granularity* of a gradual typing system defines the minimum size of such pieces in terms of abstract syntax. A performance evaluation must define its own granularity to systematically explore the ways that a programmer may write type annotations, subject to practical constraints.

Definition (*granularity*) The *granularity* of an evaluation is the syntactic unit at which the evaluation adds or removes type annotations.

For example, the evaluation in Takikawa et al. [4] is at the granularity of modules. The evaluation in Vitousek et al. [8] is at the granularity of whole programs. Section 3.2 defines the *function and class-fields* granularity, which we use for this evaluation.

After defining a granularity, a performance evaluation must define a suite of programs to measure. A potential complication is that such programs may depend on external libraries or other modules that lie outside the scope of the evaluation. It is important to distinguish these so-called *fixed modules* from the focus of the experiment.

[4]This goal is implicit in the implementation of Reticulated [8].

Definition (*experimental, fixed*) The *experimental modules* in a program define its configurations. The *fixed modules* in a program are common across all configurations.

The granularity and experimental modules define the *configurations* of a fully-typed program.

Definition (*configurations*) Let $P \rightarrow P'$ if and only if program P' can be obtained from P by annotating one syntactic unit in an experimental module. Let $\rightarrow^*$ be the reflexive, transitive closure of the $\rightarrow$ relation.[5] The *configurations* of a fully-typed program P^τ are all programs P such that $P \rightarrow^* P^\tau$. Furthermore, P^τ is a so-called *fully-typed configuration*; an *untyped configuration* P^λ has the property $P^\lambda \rightarrow^* P$ for all configurations P.

An evaluation must measure the performance overhead of these configurations relative to some default. A natural baseline is the performance of the original program, distinct from the gradual typing system.

Definition (*baseline*) The *baseline performance* of a program is its running time in the absence of gradual typing.

In Typed Racket, the baseline is the performance of Racket running the untyped configuration. In Reticulated, the baseline is Python running the untyped configuration. This is not the same as Reticulated running the untyped configuration because Reticulated inserts checks in untyped code [7].

Definition (*performance ratio*) A *performance ratio* is the running time of a configuration divided by the baseline performance of the untyped configuration.

An *exhaustive* performance evaluation measures the performance of every configuration. The natural way to interpret this data is to choose a notion of "good performance" and count the proportion of "good" configurations. In this spirit, Takikawa et al. [4] ask programmers to consider the performance overhead they could deliver to clients.

Definition (*D-deliverable*) For $D \in \mathbb{R}^+$, a configuration is *D-deliverable* if its performance ratio is no greater than D.

If an exhaustive performance evaluation is infeasible, an alternative is to select configurations via simple random sampling and measure the proportion of D-deliverable configurations in the sample. Repeating this sampling experiment yields a *simple random approximation* of the true proportion of D-deliverable configurations.

Definition (*95%-r, s-approximation*) Given r samples each containing s configurations chosen uniformly at random, a *95%-r, s-approximation* is a 95% confidence interval for the proportion of D-deliverable configurations in each sample.

The appendix contains mathematical and empirical justification for the simple random approximation method.

[5]The $\rightarrow$ relation expresses the notion of a *type conversion step* [1, 4]. The $\rightarrow^*$ relation expresses the notion of *term precision* [3].

3.2 Protocol

Granularity The evaluation presented in section 4 is at the granularity of *function and class fields*. One syntactic unit in the experiment is either one function, one method, or the collection of all fields for one class. The class in figure 1, for example, has 3 syntactic units at this granularity.

Benchmark Creation To convert a Reticulated program into a benchmark, we: (1) build a driver module that runs the program and collects timing information; (2) remove any non-determinism or I/O actions;[6] (3) partition the program into experimental and fixed modules; and (4) add type annotations to the experimental modules. We modify any Python code that Reticulated's type system cannot validate, such as code that requires untagged unions or polymorphism.

Data Collection For benchmarks with at most 2^{21} configurations, we conduct an exhaustive evaluation. For larger benchmarks we conduct a simple random approximation using ten samples each containing $10*(F+C)$ configurations, where F is the number of functions in the benchmark and C is the number of classes. *Note* the number 10 is arbitrary; our goal was to collect as much data as possible in a reasonable amount of time. *End*

All data in this paper was produced by jobs we sent to the *Karst at Indiana University*[7] computing cluster. Each job:

1. reserved all processors on one node;
2. downloaded fresh copies of Python 3.4.3 and Reticulated (commit `e478343` on the `master` branch);
3. repeatedly: selected a random configuration from a random benchmark, ran the configuration's main module 40 times, and recorded the result of each run.

Cluster nodes are IBM NeXtScale nx360 M4 servers with two Intel Xeon E5-2650 v2 8-core processors, 32 GB of RAM, and 250 GB of local disk storage.

4 Performance Evaluation

To assess the run-time cost of gradual typing in Reticulated, we measured the performance of twenty-one benchmark programs. Figure 4 tabulates information about the size and structure of the experimental portions of these benchmarks. The four columns report the lines of code (**SLOC**), number of modules (**M**), number of function and method definitions (**F**), and number of class definitions (**C**). Section 2 of the appendix describes the benchmarks' origin and purpose.

The following three subsections present the results of the evaluation. Section 4.1 reports the performance of the untyped and fully-typed configurations. Section 4.2 plots the proportion of D-deliverable configurations for D between 1 and 10. Section 4.3 compares the number of type annotations in each configuration to its performance.

[6] Four benchmarks inadvertantly perform I/O actions, see section 5.
[7] kb.iu.edu/d/bezu

Benchmark	SLOC	M	F	C
`futen`	221	3	13	2
`http2`	86	2	3	1
`slowSHA`	210	4	14	3
`call_method`	115	1	6	1
`call_simple`	113	1	6	0
`chaos`	190	1	12	3
`fannkuch`	41	1	1	0
`float`	36	1	5	1
`go`	80	1	6	1
`meteor`	100	1	8	0
`nbody`	101	1	5	0
`nqueens`	37	1	2	0
`pidigits`	33	1	5	0
`pystone`	177	1	13	1
`spectralnorm`	31	1	5	0
`Espionage`	93	2	11	1
`PythonFlow`	112	1	11	1
`take5`	130	3	14	2
`sample_fsm`	148	5	17	2
`aespython`	403	6	29	5
`stats`	1118	13	79	0

Figure 4: Static summary of benchmarks

4.1 Performance Ratios

The table in figure 5 lists the extremes of gradual typing in Reticulated. From left to right, these are: the performance of the untyped configuration relative to the Python baseline (the *retic/python ratio*), the performance of the fully-typed configuration relative to the untyped configuration (the *typed/retic ratio*), and the overall delta between fully-typed and Python (the *typed/python ratio*).

For example, the row for `futen` reports a retic/python ratio of 1.58. This means that the average time to run the untyped configuration of the `futen` benchmark using Reticulated was 1.58 times slower than the average time of running the same code using Python. Similarly, the typed/retic ratio for `futen` states that the fully-typed configuration is 1.06 times slower than the untyped configuration.

Conclusions Migrating a benchmark to Reticulated, or from untyped to fully-typed, always adds performance overhead. The migration never improves performance. The overhead is always within an order-of-magnitude. Regarding the retic/python ratios: eleven are below 2x, six are between 2x and 3x, and the remaining four are below 4.5x. The typed/retic ratios are typically lower: sixteen are below 2x, two are between 2x and 3x, and the final three are below 3.5x.

Fourteen benchmarks have larger retic/python ratios than typed/retic ratios. Given that an untyped Reticulated program offers the same safety guarantees as Python, it is surprising that the retic/python ratios are so large.

Benchmark	retic / python	typed / retic	typed / python
futen	1.58	1.06	1.68
http2	3.07	1.18	3.63
slowSHA	1.66	1.18	1.96
call_method	4.48	1.74	7.79
call_simple	1.00	3.10	3.11
chaos	2.08	1.77	3.69
fannkuch	1.14	1.01	1.15
float	2.18	1.52	3.32
go	3.77	1.97	7.44
meteor	1.56	1.37	2.13
nbody	1.78	1.01	1.80
nqueens	1.25	1.57	1.96
pidigits	1.02	1.02	1.05
pystone	1.36	2.06	2.79
spectralnorm	2.01	3.47	6.98
Espionage	2.87	1.79	5.14
PythonFlow	2.38	3.04	7.23
take5	1.21	1.14	1.38
sample_fsm	2.80	2.16	6.07
aespython	3.41	1.74	5.93
stats	1.09	1.39	1.52

Figure 5: Performance ratios

4.2 Overhead Plots

Figure 6 summarizes the overhead of gradual typing in the benchmark programs. Each plot reports the percent of D-deliverable configurations (y-axis) for values of D between 1x overhead and 10x overhead (x-axis). The x-axes are log-scaled to focus on low overheads; vertical tick marks appear at 1.2x, 1.4x, 1.6x, 1.8x, 4x, 6x, and 8x overhead.

The heading above the plot for a given benchmark states the benchmark's name and indicate whether the data is exhaustive or approximate. If the data is exhaustive, this heading lists the number of configurations in the benchmark. If the data is approximate, the heading lists the number of samples and the number of randomly-selected configurations in each sample.

Note the curves for the approximate data (i.e., the curves for `sample_fsm`, `aespython`, and `stats`) are intervals. For instance, the height of an interval at $x = 4$ is the range of the 95%-10, $[10(F + C)]$-approximation for the number of 4-deliverable configurations. These intervals are thin because there is little variance in the proportion of D-deliverable configurations across the ten samples. *End*

How to Read the Plots Overhead plots are cumulative distribution functions. As the value of D increases along the x-axis, the number of D-deliverable configurations is monotonically increasing. The important question is how many configurations are D-deliverable for low values of D. If this number is large, then a developer who applies gradual typing to a similar program has a large chance that the configuration they arrive at is a D-deliverable configuration. The area under the curve is the answer to this question. A curve with a large shaded area below it implies that a large number of configurations have low performance overhead.

The second most important aspects of an overhead plot are the two values of D where the curve starts and ends. More precisely, if $h : \mathbb{R}^+ \to \mathbb{N}$ is a function that counts the percent of D-deliverable configurations in a benchmark, the critical points are the smallest overheads d_0, d_1 such that $h(d_0) > 0\%$ and $h(d_1) = 100\%$. An ideal start-value would lie between zero and one; if $d_0 < 1$ then at least one configuration runs faster than the Python baseline. The end-value d_1 is the overhead of the slowest-running configuration.

Lastly, the slope of a curve corresponds to the likelihood that accepting a small increase in performance overhead increases the number of deliverable configurations. A flat curve (zero slope) suggests that the performance of a group of configurations is dominated by a common set of type annotations. Such observations are no help to programmers facing performance issues, but may help language designers find inefficiencies in their implementation of gradual typing.

Conclusions Curves in figure 6 typically cover a large area and reach the top of the y-axis at a low value of D. This value is always less than 10. In other words, every configuration in the experiment is 10-deliverable. For many benchmarks, the maximum overhead is significantly lower. Indeed, eight benchmarks are 2-deliverable.

None of the configurations in the experiment run faster than the Python baseline. This is to be expected, given the retic/python ratios in figure 5 and the fact that Reticulated translates type annotations into run-time checks.

Fourteen benchmarks have relatively smooth slopes. The plots for the other four benchmarks have wide, flat segments. These flat segments are due to functions that are frequently executed in the benchmarks' traces; all configurations in which one of these functions is typed incur a significant performance overhead.

Eighteen benchmarks are roughly T-deliverable, where T is the typed/python ratio listed in figure 5. In these benchmarks, the fully-typed configuration is one of the slowest configurations. The notable exception is `spectralnorm`, in which the fully-typed configuration runs faster than 38% of all configurations. Unfortunately, this speedup is due to a soundness bug;[8] in short, the implementation of Reticulated does not type-check the contents of tuples.

4.3 Absolute Running Times

Since changing the type annotations in a Reticulated program changes its performance, the language should provide a cost model to help developers predict the performance of a given configuration. The plots in figure 7 demonstrate that

[8] Bug report: github.com/mvitousek/reticulated/issues/36

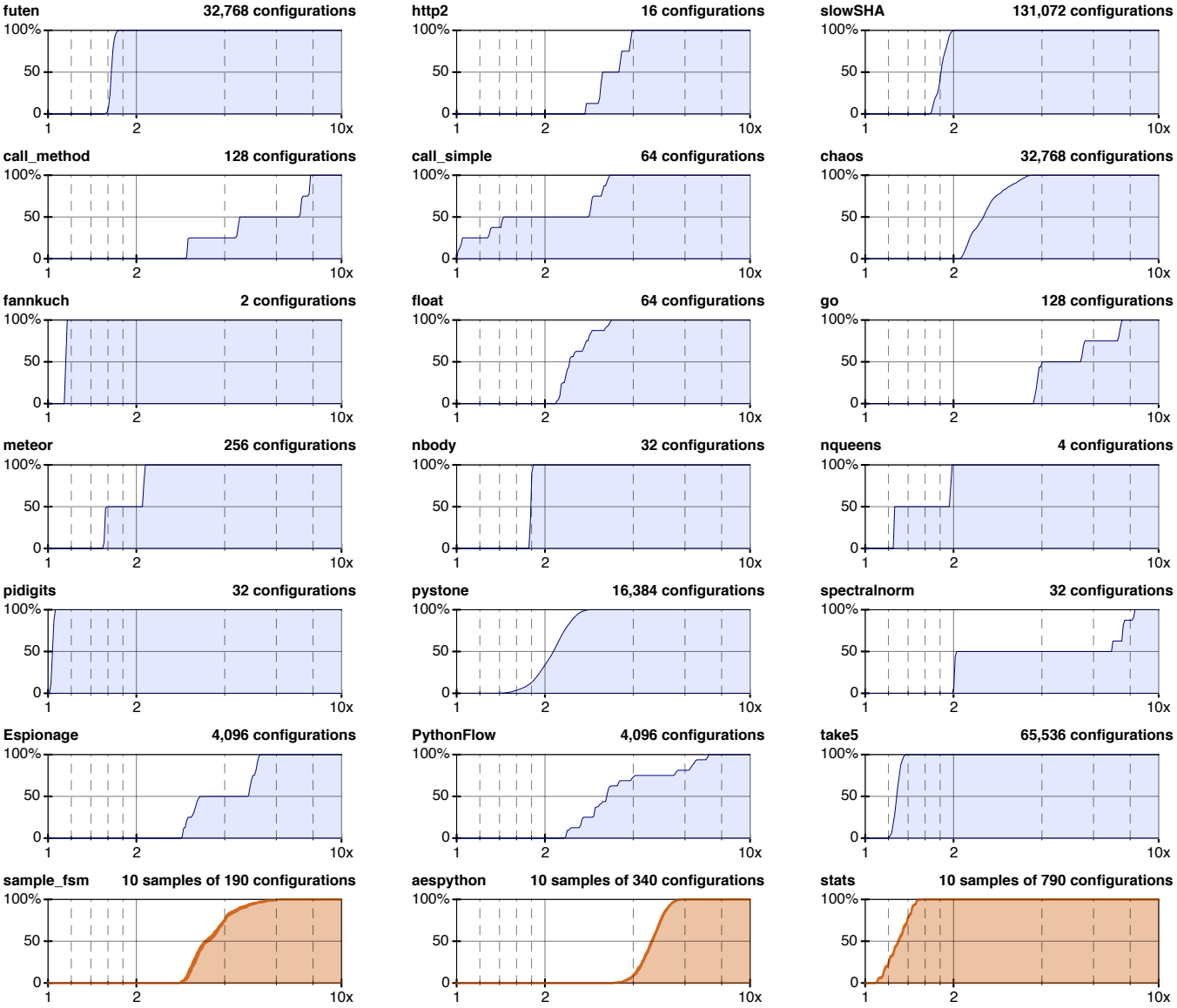

Figure 6: Overhead plots

a simple heuristic works well for these benchmarks: the performance of a configuration is proportional to the number of type annotations in the configuration.

How to Read the Plots Figure 7 contains one point for every run of every configuration in the experiment. (Recall from section 3.2, the data for each configuration is 40 runs.) Each point compares the number of type annotations in a configuration (x-axis) against its running time, measured in seconds (y-axis).

The plots contain many points with both the same number of typed components and similar performance. To reduce the visual overlap between such points, the points for a given configuration are spread across the x-axis; in particular, the 40 points for a configuration with N typed components lie within the interval $N \pm 0.4$ on the x-axis.

For example, `fannkuch` has two configurations: the untyped configuration and the fully-typed configuration. To determine whether a point (x, y) in the plot for `fannkuch` represents the untyped or fully-typed configuration, round x to the nearest integer.

Conclusions Suppose a programmer starts at an arbitrary configuration and adds some type annotations. The plots in figure 7 suggest that this action will affect performance in one of four possible ways, based on trends among the plots.

Trend I *(types make things slow)*: The plots for ten benchmarks show a gradual increase in performance overhead

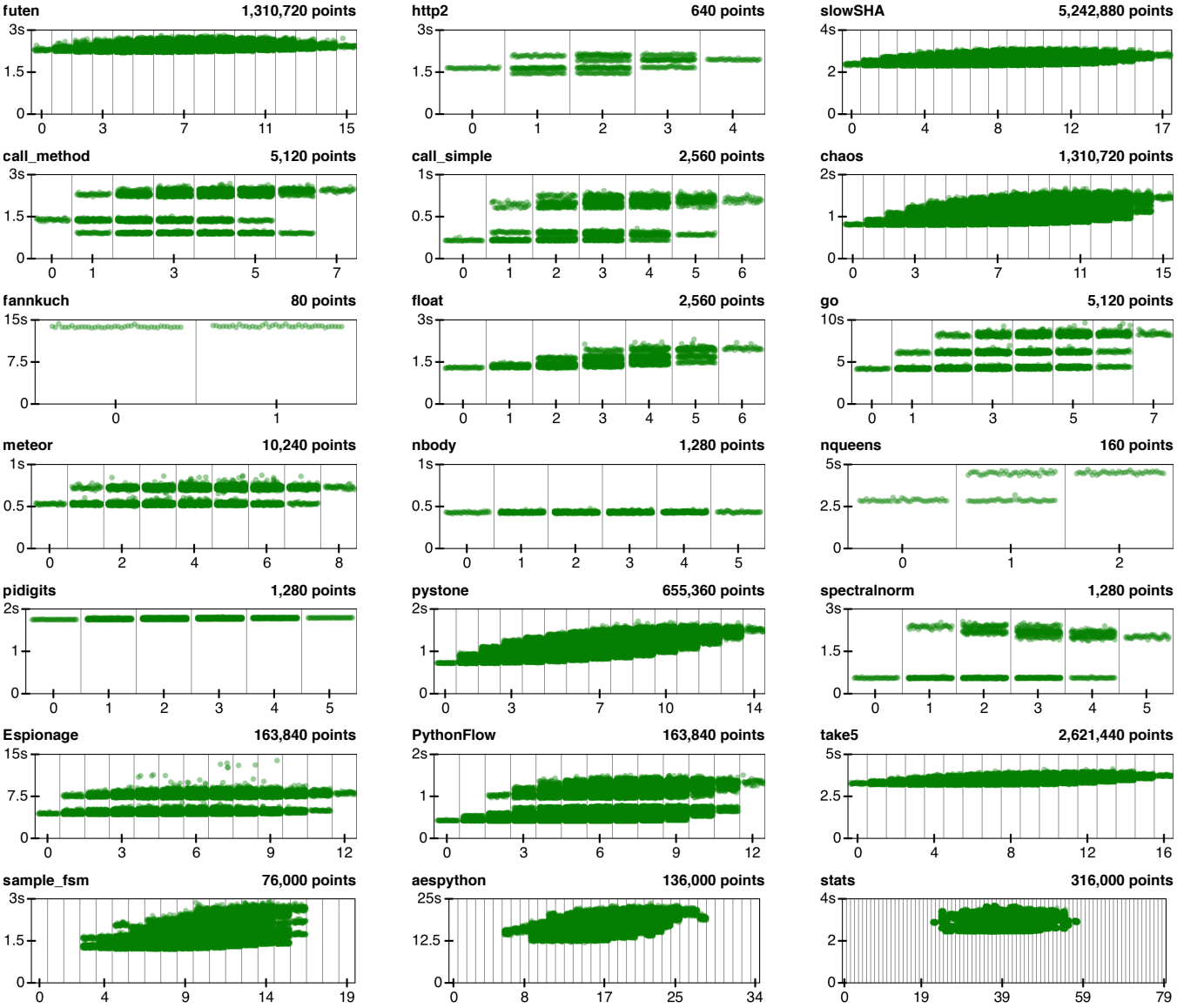

Figure 7: Running time (in seconds) vs. Number of typed components

as the number of typed components increases. Typing any function, class, or method adds a small performance overhead. Applies to `futen`, `slowSHA`, `chaos`, `float`, `pystone`, `PythonFlow`, `take5`, `sample_fsm`, `aespython`, and `stats`.

Trend II *(types make things very slow)*: Nine plots have visible gaps between clusters of configurations with the same number of types. Configurations below the gap contain type annotations that impose relatively little run-time cost. Configurations above the gap have some common type annotations that add significant overhead. Each such gap corresponds to a flat slope in figure 6. Applies to `call_method`, `call_simple`, `go`, `http2`, `meteor`, `nqueens`, `spectralnorm`, `Espionage`, and `PythonFlow`.

Trend III *(types are free)*: In three benchmarks, all configurations have similar performance. The dynamic checks that enforce tag soundness add insignificant overhead. Applies to `fannkuch`, `nbody`, and `pidigits`.

Trend IV *(types make things fast)*: In two benchmarks, some configurations run faster than similar configurations with fewer typed components. These speedups are the result of two implementation bugs: (1) Reticulated does not dynamically check the contents of statically-typed tuples, and (2) for method calls to dynamically-typed objects, Reticulated performs a run-time check that overlaps with Python's dynamic typing [7]. Applies to `call_method` and `spectralnorm`.

Overall, there is a clear trend that adding type annotations adds performance overhead. The increase is typically linear. On one hand, this observation may help programmers predict performance issues. On the other hand, the linear increase demonstrates that Reticulated does not use type information to optimize programs. In principle a JIT compiler could generate check-free code if it could infer the run-time type of a variable, but it remains to be seen whether this approach would improve performance in practice.

5 Threats to Validity

We have identified five sources of systematic bias. First, the experiment consists of a small suite of benchmarks, and these benchmarks are rather small. For example, an ad-hoc sample of the PyPI Ranking[9] reveals that even small Python packages have far more functions and methods than our benchmarks. The `simplejson` library contains over 50 functions and methods, the `requests` library contains over 200, and the `Jinja2` library contains over 600.

Second, the experiment considers one fully-typed configuration per benchmark; however, there are many ways of typing a given program. The types in this experiment may differ from types ascribed by another Python programmer, which, in turn, may lead to different performance overhead.

Third, some benchmarks use dynamic typing. The `take5` benchmark contains one function that accepts optional arguments, and is therefore dynamically typed.[10] The `go` benchmark uses dynamic typing because Reticulated cannot validate its use of a recursive class definition. The `pystone` and `stats` benchmarks use dynamic typing to overcome Reticulated's lack of untagged union types.

Fourth, the `aespython`, `futen`, `http2`, and `slowSHA` benchmarks read from a file within their timed computation. We nevertheless consider our results representative.

Fifth, Reticulated supports a finer granularity of type annotations than the experiment considers. Function signatures can leave some arguments untyped, and class field declarations can omit types for some members. We believe that a fine-grained evaluation would support the conclusions presented in this paper.

6 Is Sound Gradual Typing Alive?

Our application of the Takikawa method suggests that any combination of statically typed and dynamically typed code in Reticulated runs within one order of magnitude of the original Python program. This relatively impressive performance comes at a three-fold cost. First, soundness is at the level of type-tags rather than full static types. Second, run-time tag errors do not describe the source of the ill-typed value. Third, fully-typed programs typically suffer more overhead than any other combination of typed and untyped code.

Our evaluation thus raises a number of open research problems. First among these is whether programmers will find the static guarantees of tag soundness useful for maintaining large programs. In our experience, well-tagged programs often contain subtle mistakes.

A second question is how the cost of soundness compares to the cost of expressive types and precise error messages. Experience by Vitousek et al. [8] suggests that the cost of useful error messages is high. They extend Reticulated to track a set of possibly-guilty boundaries and find that maintaining the set doubled the typed/retic ratio in the majority of their benchmark programs.

A third question is whether Reticulated can reduce its overhead relative to Python. Ideally, untyped Reticulated programs should have the same performance as Python.

Finally, we ask whether Reticulated can leverage type information to remove run-time checks from Python programs. The current implementation performs far worse than Typed Racket on fully-typed programs because the latter only adds run-time checks at boundaries between statically-typed and dynamically-typed code.

Appendix

1 Validating the Approximation Method

Section 3 proposes a so-called *simple random approximation* method for guessing the number of D-deliverable configurations in a benchmark:

Definition (95%-r, s-*approximation*) Given r samples each containing s configurations chosen uniformly at random, a 95%-r, s-*approximation* is a 95% confidence interval for the proportion of D-deliverable configurations in each sample.

Section 4 instantiates this method using $r = 10$ samples each containing $10 * (F + C)$ configurations, where F is the number of functions and methods in the benchmark and C is the number of class definitions. The intervals produced by this method (for the `sample_fsm`, `aespython`, and `stats` benchmarks) are thin, but the paper does not argue that the intervals are very likely to be accurate. This appendix provides the missing argument.

1.1 Statistical Argument

Let d be a predicate that checks whether a configuration from a fixed program is D-deliverable. Since d is either true or false for every configuration, this predicate defines a Bernoulli random variable X_d with parameter p, where p is the true proportion of D-deliverable configurations. Consequently, the expected value of this random variable is p. The law of large numbers therefore states that the average of infinitely many samples of X_d converges to p, the true proportion of deliverable configurations. Convergence suggests that the average of "enough" samples is "close to" p. The central limit theorem provides a similar guarantee—any sequence of such

[9] pypi-ranking.info/alltime
[10] Bug report: github.com/mvitousek/reticulated/issues/32.

averages is normally distributed around the true proportion. A 95% confidence interval generated from sample averages is therefore likely to contain the true proportion.

1.2 Empirical Illustration

Figure 8 superimposes the results of simple random sampling upon the exhaustive data for three benchmarks. Specifically, these plots are the result of a two-step recipe:

- First, we plot the true proportion of D-deliverable configurations for D between 1x and 10x. This data is represented by a blue curve; the area under the curve is shaded blue.
- Second, we plot a 95%-10, $[10(F + C)]$-approximation as a brown interval. This is a 95% confidence interval generated from ten samples each containing $10(F+C)$ configurations chosen uniformly at random.

2 Benchmark Descriptions

Five benchmarks originate from case studies by Vitousek et al. [7]. Twelve are from the evaluation by Vitousek et al. [8] on programs from the Python Performance Benchmark Suite. The remaining four originate from open-source programs.

The following descriptions credit each benchmark's original author, state whether the benchmark depends on any fixed modules, and briefly summarize its purpose.

futen from `momijiame`
Depends on the `fnmatch`, `os.path`, `re`, `shlex`, and `socket` libraries.
Converts an OpenSSH configuration file to an inventory file for the *Ansiable* automation framework.

http2 from Joe Gregorio
Depends on the `urllib` library.
Converts a collection of Internationalized Resource Identifiers to equivalent ASCII resource identifiers.

slowSHA from Stefano Palazzo
Depends on the `os` library.
Applies the SHA-1 and SHA-512 algorithms to English words.

call_method from The Python Benchmark Suite
No dependencies.
Microbenchmarks simple method calls; the calls do not use argument lists, keyword arguments, or tuple unpacking.

call_simple from The Python Benchmark Suite
No dependencies.
Same as `call_method`, using functions rather than methods.

chaos from The Python Benchmark Suite
Depends on the `math` and `random` libraries.
Creates fractals using the *chaos game* method.

fannkuch from The Python Benchmark Suite
No dependencies.
Implements Anderson and Rettig's microbenchmark.

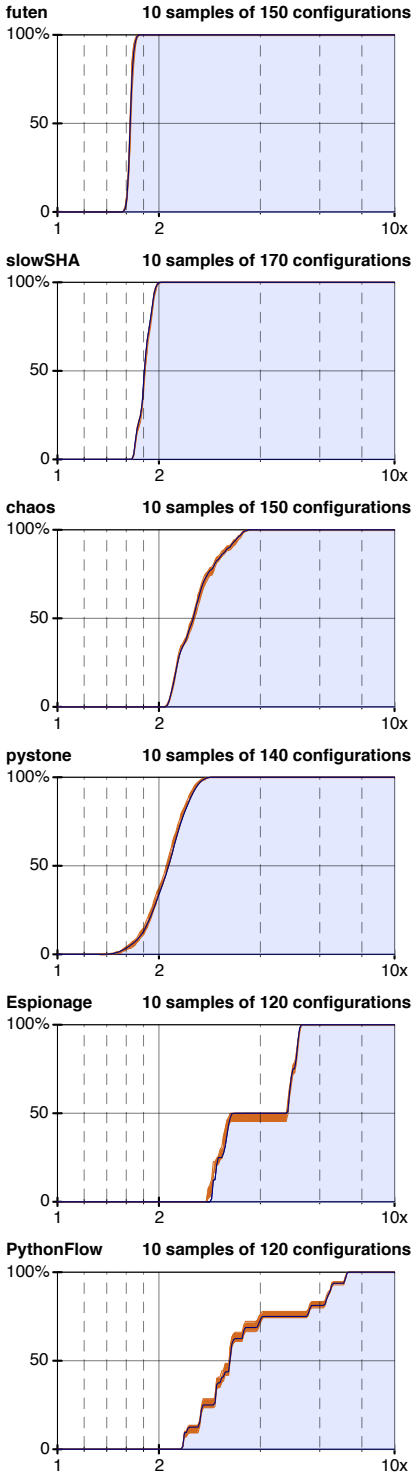

Figure 8: Simple random approximations

float from The Python Benchmark Suite
Depends on the `math` library.
Microbenchmarks floating-point operations.

go from The Python Benchmark Suite
Depends on the `math` and `random` libraries, and two untyped modules.
Implements the game Go. This benchmark is split across three files: an experimental module that implements the game board, a fixed module that defines constants, and a fixed module that implements an AI and drives the benchmark.

meteor from The Python Benchmark Suite
No dependencies.
Solves the Shootout benchmarks meteor puzzle. [11]

nbody from The Python Benchmark Suite
No dependencies.
Models the orbits of Jupiter, Saturn, Uranus, and Neptune.

nqueens from The Python Benchmark Suite
No dependencies.
Solves the 8-queens problem by a brute-force algorithm.

pidigits from The Python Benchmark Suite
No dependencies.
Microbenchmarks big-integer arithmetic.

pystone from The Python Benchmark Suite
No dependencies.
Implements Weicker's *Dhrystone* benchmark. [12]

spectralnorm from The Python Benchmark Suite
No dependencies.
Computes the largest singular value of an infinite matrix.

Espionage from Zeina Migeed
Depends on the `operator` library.
Implements Kruskal's spanning-tree algorithm.

PythonFlow from Alfian Ramadhan
Depends on the `os` library.
Implements the Ford-Fulkerson max flow algorithm.

take5 from Maha Alkhairy and Zeina Migeed
Depends on the `random` and `copy` libraries.
Implements a card game and a simple player AI.

sample_fsm from Linh Chi Nguyen
Depends on the `itertools`, `os`, and `random` libraries.
Simulates the interactions of economic agents modeled as finite-state automata.

aespython from Adam Newman and Demur Remud
Depends on the `os` and `struct` libraries.
Implements the Advanced Encryption Standard.

stats from Gary Strangman
Depends on the `copy` and `math` libraries.
Implements first-order statistics functions; in other words, transformations on either floats or (possibly-nested) lists of floats. The original program consists of two modules. The benchmark is modularized according to comments in the program's source code to reduce the size of each module's configuration space.

[11] benchmarksgame.alioth.debian.org/u32/meteor-description.html
[12] eembc.org/techlit/datasheets/ECLDhrystoneWhitePaper2.pdf

Acknowledgments

This paper is supported by NSF grant CCF-1518844. Part of this work was completed while the second author was an REU under Jeremy Siek at Indiana University. We thank Spenser Bauman, Matthias Felleisen, Tony Garnock-Jones, Sam Tobin-Hochstadt, Michael Vitousek, Ming-Ho Yee, and the PEPM review committee.

References

[1] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to Evaluate the Performance of Gradual Type Systems. Submitted for publication, 2017.

[2] Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2017.

[3] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In *Proc. Summit oN Advances in Programming Languages*, 2015.

[4] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 456–468, 2016.

[5] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 395–406, 2008.

[6] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten years later. In *Proc. Summit oN Advances in Programming Languages*, 2017.

[7] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. Dynamic Languages Symposium*, pp. 45–56, 2014.

[8] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *Proc. ACM Symposium on Principles of Programming Languages*, 2017.

[9] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *Proc. European Conference on Object-Oriented Programming*, pp. 28:1–28:29, 2017.