On the Cost of Soundness for Gradual Typing

Ben Greenman
Northeastern University
benjaminlgreenman@gmail.com

Zeina Migeed
Northeastern University
migeed.z@outlook.com

Abstract

Gradual typing promises to reduce the cost of software maintenance for dynamically typed languages. In a language with a gradual typing system, developers can add type annotations to a portion of a code base after they reconstruct its type during some maintenance action. As Takikawa et al.'s recent work shows, however, the addition of type annotations comes at a large cost in performance. In particular, performance evaluations of Typed Racket suggest that a conventionally sound gradual typing system may slow down a working system 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 in contrast to Typed Racket, satisfies a more relaxed notion of soundness which we refer to as tag soundness. We find that the cost of soundness in Reticulated is at most one order of magnitude. Substantial user studies are needed to determine whether programmers will accept tag soundness.

ACM Reference Format:
Ben Greenman and Zeina Migeed. 2017. On the Cost of Soundness for Gradual Typing. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 How Much does Gradual Soundness Cost?

Gradual typing systems can help programmers with the task of maintaining code written in a dynamically typed language. If the latter 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 piece of the code base can use the type annotations to understand its structure and invariants.

Conference'17, July 2017, Washington, DC, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While gradual typing can improve readability and robustness, it has serious implications for performance [2, 4]. The problem is that gradual typing systems implement soundness with run-time type checks, and 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. An evaluation by Takikawa et al. measured the performance of Typed Racket's generalized type soundness [6]. In this paper, we apply the same method to measure Reticulated Python's tag soundness [8], which is a more relaxed notion of soundness.

Tag soundness guarantees that if a well-typed expression reduces to a value, then the value and expression share the same top-level type constructor (see section 2.1). Thus an expression with type List(Int) can reduce to a list of strings, but not to an integer or a function.

In contrast, generalized type soundness guarantees that if an expression with type τ reduces to a value, then the value is well-typed at τ. Furthermore, if the evalution ends in a type error, generalized soundness guarantees that the error points to one of the finitely-many boundaries between statically-typed and dynamically-typed code, thereby helping programmers diagnose the source of the error.

Generalized soundness clearly provides stronger guarantees than tag soundness, but Takikawa et al. [4] show that its implementation in Typed Racket can slow programs by two orders of magnitude. Prior work on Reticulated does not report the performance of gradually-typed programs [7, 8].

This paper reports on the application of the Takikawa performance evaluation method [4] to Reticulated. The central findings are that:

- Reticulated experiences a slow down of at most one order of magnitude.
- The performance degradation is approximately a linear function of the number of type annotations.
- Despite the smaller performance overhead, it remains to be seen whether developers will tolerate the overhead of gradual typing in Reticulated and the weaker notion of soundness.

To set the stage, this paper first describes the point that Reticulated occupies in the design space of gradual typing systems (section 2). It then explains our adaptation of the Takikawa method to Reticulated (section 3). Section 4 presents the details of the evaluation: the benchmarks, the measurements, and the conclusions.

```
@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

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`. These defensive checks protect the statically typed method from arbitrary callers. If a Python context invokes `add_cash` with an integer, the inserted checks will halt the program with a so-called *dynamic type error*.

2.1 Tag Soundness

Reticulated uses dynamic type checks to implement a form of type soundness.[2] Informally, if `e` is a well-typed expression, then evaluating `e` can result in four possible 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);
4. the execution ends in a dynamic type 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.[3]

Tag soundness is weaker than standard type soundness in two ways. First, tag soundness does not rule out type errors in well-typed programs. Second, tag soundness implies that an expression with type `List(Int)` can produce any kind of `List`. In figure 3, for example, the expression `make_ints()` has the static type `List(Int)` but evaluates to a list containing a string, an empty list, and a function. Put another way, Reticulated supports only tag-level compositional reasoning.

[1]Specifically, CPython 3.

[2]Vitousek et al. [8] use the phrase *open-world soundness*. They conjecture that Reticulated is open-world sound.

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

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

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

$$\begin{array}{ll} \lfloor\mathsf{Int}\rfloor = \mathsf{Int} & \lfloor\mathsf{Function}([\tau],\tau')\rfloor = \mathsf{Function} \\ \lfloor\mathsf{List}(\tau)\rfloor = \mathsf{List} & \lfloor\mathsf{Dyn}\rfloor = \mathsf{Dyn} \end{array}$$

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

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

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

Figure 3: A strange but well-typed function

Nevertheless, tag soundness is a very useful guarantee in the context of Reticulated. Reticulated's main design goal is to provide seamless interaction with Python code. To quote the vision paper of Siek et al. [3]:

> *[P]rogrammers should be able to add or remove type annotations without any unexpected impacts on their program, such as whether it still type-checks and whether its runtime behavior remains the same.*

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 dynamic type 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 a static analysis of the Python code. Tag soundness admits this reality with its fourth clause, which states that execution may end in a nondescript type error.

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 is to use the same value in both cases.[4] 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 an implicit design goal; *all dynamic type checks run in near-constant time*.[5] 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 a $\Theta(f)$ 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. Takikawa et al. 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. 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 consider 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.

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

The granularity and experiemental modules define the so-called 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.[6] 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* is a P^λ such that $P^\lambda \rightarrow^* P$ for all configurations P.

A performance evaluation must measure the running time of these configurations relative to the *baseline* performance of the untyped configuration in the absence of gradual typing. In Typed Racket, the baseline is the performance of Racket running the untyped configuration. Reticulated adds type checks to un-annotated programs [7], so its baseline is Python running the untyped configurations.

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, one 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.

Our technical appendix contains theoretical and empirical justification for the simple random approximation method [1].

[4]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).

[5]This goal is implicit in the implementation of Reticulated, and assumed by Vitousek et al. [8].

[6]The $\rightarrow$ relation expresses the notion of a *type conversion step* [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.

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;[7] (3) partition the program into experimental and control 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 a larger benchmark, with F functions and C classes, we conduct a simple random approximation using ten samples each containing $10 * (F + C)$ configurations.

All data in this paper was produced by jobs we sent to the *Karst at Indiana University*[8] high-throughput 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 to measure, 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 the 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**). Our technical appendix describes the benchmarks' origin and purpose [1].

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 demonstrates that the number of type annotations in a configuration is correlated to its performance.

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 On one hand, these ratios demonstrate that migrating a benchmark to Reticulated, or from untyped to fully-typed, always adds performance overhead. The migration never improves performance. On the other hand, 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. It is surprising that running a Python program through Reticulated causes such a large slowdown.

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

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 the nature of the underlying dataset. 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.

Technical Note: 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.

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 better channe of arriving at a D-deliverable configuration. The area under the curve is the answer; in short, more is better. 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 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 in the benchmark.

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 no surprise 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. In fact, these flat segments are due to functions that are frequently executed in the benchmarks' traces.

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; in short, the implementation of Reticulated does not type-check the contents of tuples.[9]

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

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

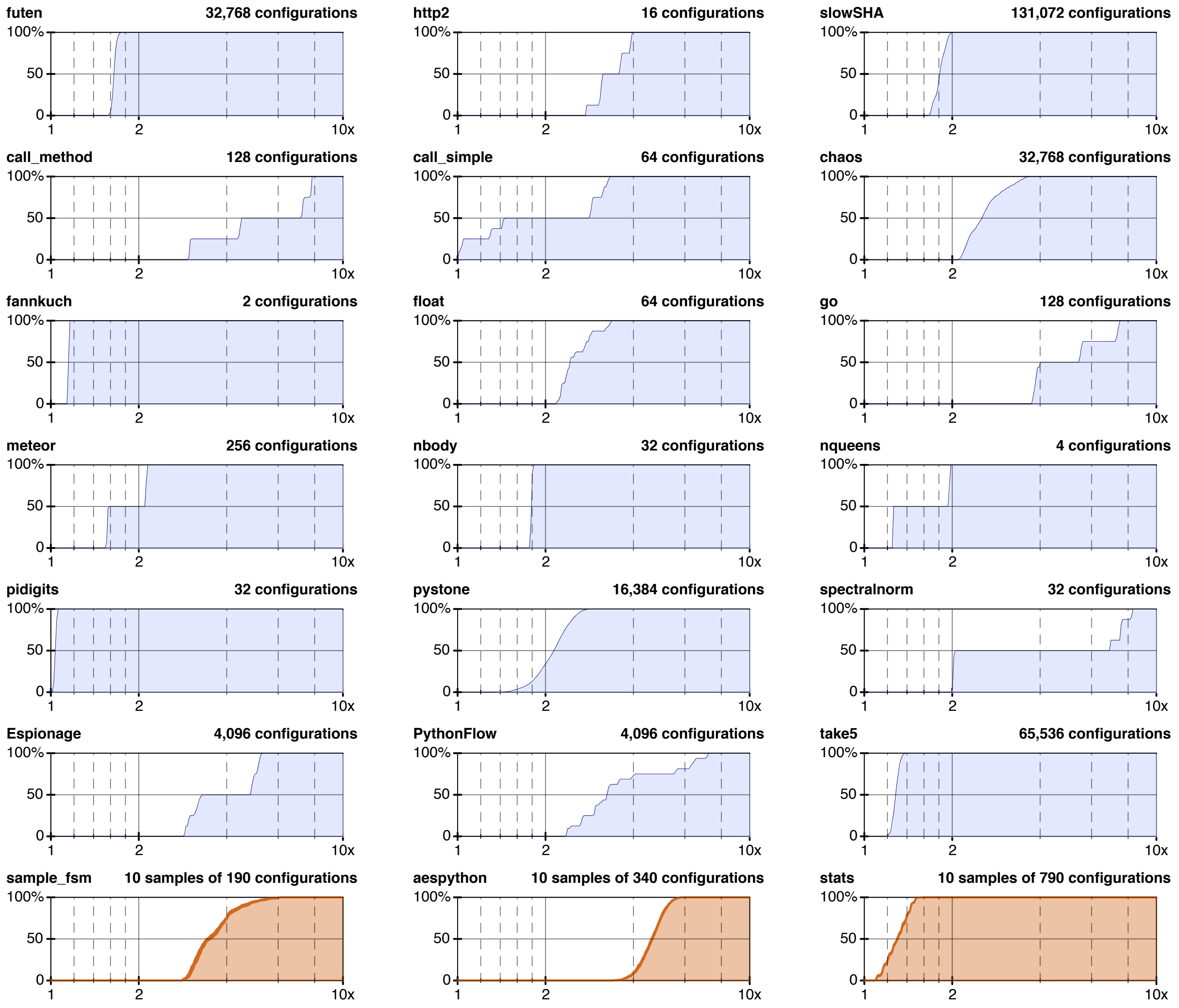

Figure 6: Overhead plots

How to Read the Plots Figure 7 contains one point for every run of every configuration in the experiment.[10] Each point compares the number of typed functions, methods, and classes 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.

[10]Recall from section 3.2, the data for each configuration is 40 runs.

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 as the number of typed components increases. Typing any function, class, or method adds a small performance overhead.

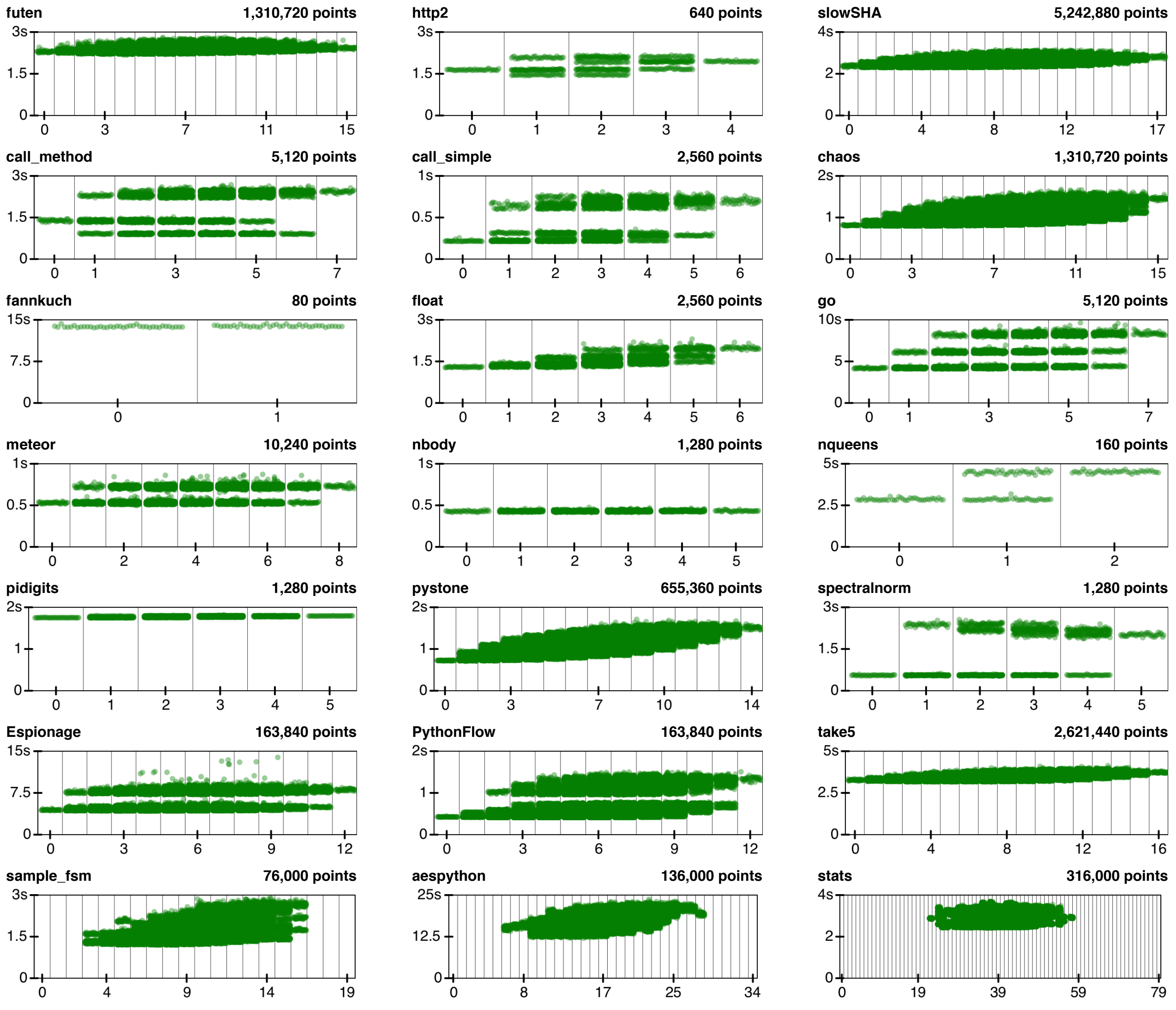

Figure 7: Running time (in seconds) vs. Number of typed components

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.

Trend III *(types are free)*: In three benchmarks, all configurations have similar performance. The dynamic checks that enforce type soundness add insignificant overhead.

Trend IV *(types make things fast)*: In two benchmarks, there are some configurations that run faster than similar configurations with fewer typed components. These speedups happen for two reasons: either because of duplicate checks on dynamically-typed receivers of method calls, or because of omitted checks on values annotated with tuple types. The former is due to an overlap between Reticulated's semantics and Python's dynamic typing [7]. The latter is due to the implementation bug noted in section 4.2.

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 that cast doubt upon the validity of our conclusions. 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[11] reveals that even small packages have far more functions and methods. 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.[12] 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. Partially-typed functions may come with entirely different performance. We leave this as an open question.

6 Is Sound Gradual Typing Alive?

The 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 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 type errors point to a set of potentially-guilty type boundaries rather than a single location.[13] Third, fully-typed programs typically suffer more overhead than any other configuration.

Our evaluation effort thus leaves us with a number of open research problems:

- Will programmers accept tag soundness? Substantial user studies are needed.
- How does the cost of soundness compare to the cost of expressive types and informative error messages? This question demands a two-pronged answer: (1) Reticulated must improve its types and error messages; (2) Typed Racket must implement a form of tag soundness.
- Can Reticulated reduce its overhead relative to Python? Ideally, Reticulated programs with no type annotations should have the same performance as Python.
- Can Reticulated use type information to remove dynamic checks from Python programs? At present, Typed Racket is more performant than Reticulated on fully-typed programs because it adds run-time checks only when linked to dynamically-typed code.

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 Matthias Felleisen, Michael Vitousek, Sam Tobin-Hochstadt, Spenser Bauman, Tony Garnock-Jones, and Ming-Ho Yee.

Bibliography

[1] Ben Greenman and Zeina Migeed. On the Cost of Soundness for Gradual Typing. Northeastern University, NU-CCIS-2017-001, 2017.

[2] 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.

[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*, pp. 274–293, 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*, pp. 762–774, 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.

[11] pypi-ranking.info/alltime

[12] Bug report: github.com/mvitousek/reticulated/issues/32.

[13] The version of Reticulated in this paper always reports an empty set. Vitousek et al. [8] improve the error messages and report that the improvement doubled the typed/retic ratio in most of their benchmark programs.