Is Sound Gradual Typing Alive?

Ben Greenman
Northeastern University
benjaminlgreenman@gmail.com

Zeina Migeed
Northeastern University
migeed.z@outlook.com

Abstract

Gradual typing empowers developers to freely combine dynamically and statically typed code in a single program. A sound gradual typing system performs run-time checks to ensure the integrity of types at the boundary between typed and untyped code. The question is how much such checks affect the performance of gradually typed software systems.

This paper presents a systematic performance evaluation of Reticulated, a gradual typing system for Python. Since Reticulated allows fine-grained mixing of typed and untyped code, our evaluation method is to establish a fixed-size unit for adding type annotations and measure the performance of all possible combinations of typed and untyped code. The paper also shows that this exponentially expensive performance evaluation method can be approximated with a linear sampling technique. In comparison to the performance of Typed Racket—the first gradual typing system to be evaluated in a comprehensive manner—the measurements look encouraging. On closer inspection, the low cost of run-time checks seem to be due to Reticulated's inexpressive type system, miserable error messages, and alternative notion of type soundness.

ACM Reference format:
Ben Greenman and Zeina Migeed. 2016. Is Sound Gradual Typing Alive?. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

Acknowledgments

This paper is supported by NSF grant CCF-1518844.

1 Measuring Reticulated

Siek and Taha's 2006 paper [6] introduces the notion of gradual typing, a sound variant of Common Lisp's optional typing. Using gradual typing, "programmers should be able to add or remove type annotations without any unexpected impacts on their program" [7]. One unexpected impact is the application of a function on integers to a string; similarly, when typed code passes an array of floats to untyped code, the programmer expects that the untyped code does not assign a boolean to one of the array's slots. By contrast, a programmer may expect the addition of types to speed up the program execution because the compiler can exploit the type information.

Gradual typing implements the sound combination of typed and untyped code with the insertion of run-time checks. At a high level, a gradual typing system uses the type annotations to determine whether and where a typed value may flow into the function position of an application and adds a check there to ensure the argument value matches the expected type. If not, the run-time check raises an exception. Clearly, the insertion of such run-time checks imposes a cost, and the question arises how significant this cost is.

Takikawa et al. [8] present the first comprehensive method for evaluating the performance of gradual typing systems. If a program consists of n components, Takikawa et al. propose to measure 2^n configurations. Doing so mimics the process through which programmers may gradually equip all possible sites for type declarations with annotations. In order to evaluate these measurements, they propose a simple metric, the proportion of *D-deliverable* configurations, meaning the number of configurations whose running time is at most Dx slower than the untyped configuration. Follow-up work by Greenman et al. [4] confirms that a *linear* amount of random sampling approximates the exponential measurements well in the case of Typed Racket.

Simultaneously to Siek and Taha's original work, Tobin-Hochstadt and Felleisen launched the development of Typed Racket, based on almost the same idea as gradual typing [9]. In 2016, with Typed Racket under development for 10 years, Takikawa et al.'s performance evaluation of Typed Racket indicates that this form of gradual typing has disastrous performance characteristics.

This paper contributes three insights. First, it explains an adaptation of Takikawa et al.'s method (see section 3) for Reticulated [11], a gradual typing system that is relatively faithful to Siek and Taha's original proposal. Second, it reports on the results of applying the adapted method to Reticulated (see section 4). While the performance of Reticulated seems to be significantly better than Typed Racket's, we conjecture that this due to (1) a significant gap in Reticulated's soundness, (2) Reticulated's lack of expressiveness, and (3) low-quality error messages (see section 7). Third, the paper confirms Greenman et al.'s insight that linear sampling is a sufficiently good approximation of the exponential set of

Conference'17, Washington, DC, USA
© 2016 ACM. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

measurements for the adapted method (see section 5). The next section presents the background material.

2 Reticulated Python

Reticulated is a gradual typing system for Python [11]. In Reticulated, programmers can express types using Python's syntax for function annotations and decorators. Reticulated statically checks the annotations and translates them to run-time type checks designed to enforce type soundness.

In a statically typed language, type soundness implies that if a program is well-typed, running the program results in one of three possible outcomes: the program (1) evaluates to a value of the expected type, (2) diverges, or (3) throws an exception from a well-defined set. For partially-typed programs, in which typed and untyped code interact, there is a fourth outcome: the program can (4) throw an exception due to a boundary between typed and untyped code [10]. Reticulated aims to enforce the first three conditions by static type-checking, and the fourth by dynamic type-checking.

Figure 1 presents a well-typed class definition. If we add the method call `c1.add_cash(20)` to the program, then Reticulated raises a static type error because the integer `20` is not an instance of the `Cash` class. Contrast this to an ill-typed call that occurs in a dynamically-typed context:

```
def dyn_add_cash(amount):
  c1.add_cash(amount)

dyn_add_cash(20)
```

The variable `amount` is not annotated, so Reticulated cannot statically prove that all calls to `dyn_add_cash` violate the assumptions of the `add_cash` method. To preserve type soundness, Reticulated rewrites the method to defensively check its arguments; in particular, Reticulated adds one structural type check for each argument of `add_cash`. At run-time, the check for the second argument throws an exception before the call `dyn_add_cash(c1, 20)` causes the program to fail.

Reticulated inserts similar checks around function calls, to enforce the declared return type, and around reads from variables or data structures, to detect strong updates [11]. These pervasive checks implement a notion of soundness that protects typed code without inhibiting untyped code [12].

```
import math

@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

c1 = Cash(1000, 0)
```

Figure 1: Reticulated syntax

3 Evaluation Method

Takikawa et al. [8] introduce a method for evaluating the performance of a gradual typing system. The method is based on the premise that a performance evaluation cannot assume how developers will apply gradual typing, nor can it assume that all developers have identical performance requirements. Therefore, the method considers all possible configurations that a developer may obtain by incrementally adding types and reports the overhead of all configurations relative to the original, untyped program.

Takikawa et al. apply this method to Typed Racket, a gradual typing system that permits typed and untyped modules. Thus a fully-typed program with M modules defines a space of 2^M configurations. Takikawa et al. measure the overhead of these configurations relative to the fully-untyped configuration and plot how the proportion of so-called *D-deliverable* configurations varies as developers instantiate the parameter D with the highest performance overhead they can tolerate.

3.1 Adapting Takikawa et al.'s Method

Reticulated supports fine-grained combinations of typed and untyped code. It would be impractical to directly apply the Takikawa method; measuring all configurations would take more time than the universe has left. It would also be impractical to ignore the fine granularity of Reticulated and apply the module-level protocol that Takikawa et al. used for Typed Racket. The practical choice lies somewhere in between, and it depends on the size of the programs at hand and computing resources available.

Definition (*granuarity*) 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. [8] is at the granularity of modules. The evaluation in Vitousek et al. [12] is at the granularity of whole programs. Section 3.2 defines the *function and class-fields* granularity used in this paper.

Once the granularity is fixed, the second step is to ascribe types to representative programs. A potential complication is that such programs may depend on external libraries or other modules that lie outside the scope of the evalutation.

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

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

Definition (*configurations*) Let $P \rightarrow_1 P'$ if and only if program P' can be obtained from P by annotating one syntactic unit in an experimental module. Let $\sqsubseteq$ be the reflexive, transitive closure of the $\rightarrow_1$ relation.[1] The *configurations* of a fully-typed program P^τ are all programs P such that $P \sqsubseteq P^\tau$.

The next step is to measure the performance of these configurations and report their overhead relative to the performance a developer would get by opting out of gradual typing. In Typed Racket, this baseline is the performance of Racket running the untyped configuration. In Reticulated, untyped variables still require run-time checks, so the baseline is the performance of Python running the untyped configuration.

Definition (*performance ratio*) A *performance ratio* is the running time of a program divided by the running time of the same program in the absence of gradual typing.

After measuring the performance ratio of each configuration, the final step is to classify configurations by their performance. Since different applications have different performance requirements, the only rational way to report performance is with a parameterized metric.

Definition (*D-deliverable*) For real-valued D, the proportion of D-deliverable configurations is the proportion of configurations with performance ratios no greater than D.

3.2 Protocol

Sections 4 and 5 report the performance of Reticulated at a function and class-fields granularity; more precisely, one syntactic unit in the experiment is either one function, one method, or the collection of all fields for one class. The evaluation furthermore adheres to the following protocols for benchmark creation and data collection.

Benchmark Creation Given a Python program, we first build a driver module that performs some non-trivial computation using the program. Second, we remove any nondeterminism or I/O actions from the program. Third, we define the experimental modules. Fourth, we ascribe types to the experimental modules.

When possible, we use existing type annotations or comments to infer the fully-typed configuration. When necessary, we use details of the driver module to infer types; for example, any polymorphic functions must use monomorphic types because Reticulated does not support polymorphism.

Data Collection We enumerate the configuration space and choose a random permutation of the enumeration. Optionally, we divide the permutation across identical processors or machines. We run the main module of the configuration a fixed number of times and record each running time. Finally, we run the main module of the untyped configuration using the Python 3.4.3 interpreter.[2]

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_method_slots`	116	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

Figure 2: Static summary of benchmarks

Details of the Evaluation All data in this paper was produced by jobs we sent to the *Karst at Indiana University*[3] high-throughput computing cluster. Each job:

1. reserved all processors on one node for 24 hours;
2. downloaded fresh copies of Python 3.4.3 and Reticulated (commit `e478343` on the `master` branch);
3. for the rest of the 24-hour span: 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.

Three details of the Karst protocol warrant further attention. First, nodes selected a random configuration by reading from a text file that contained a permutation of the configuration space. This text file was stored on a dedicated machine. Second, the same dedicated machine that stored the text file also stored all configurations for each *module* of each benchmark. After a node selected a configuration to run, it copied the relevant files to private storage before running the main module. Third, we wrapped the main computation of every benchmark in a `with` statement that recorded execution time using the Python function `time.process_time()`.

[1]The $\rightarrow_1$ relation expresses the notion of a *type conversion step* [8]. The $\sqsubseteq$ relation expresses the notion of *term precision* [7].

4 Exhaustive Evaluation

This section presents the results of an exhaustive performance evaluation of nineteen benchmark programs. The benchmarks are small Python programs whose implicit types are expressible in Reticulated. The results are performance ratios (figure 3), overhead plots (figure 4), and a series of graphs comparing the number of typed components in a configuration against the configuration's performance (figure 5).

4.1 About the Benchmarks

Of the nineteen benchmark programs, three originate from case studies by Vitousek et al. [11], thirteen are from the evaluation by Vitousek et al. [12] on programs from the Python Performance Benchmark Suite, and the remaining three originate from open-source programs. Every listing of the benchmarks in this section is ordered first by the benchmark's origin and second by the benchmark's name.

Figure 2 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**).

The following descriptions credit each benchmark's original author, state whether it depends on any control 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* 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_method_slots from The Python Benchmark Suite
No dependencies.
Same as `call_method`, but using receiver objects that declare their methods in their `__slots__` attribute.

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 [1].

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 control module that defines constants, and a control module that implements an AI and drives the benchmark.

meteor from The Python Benchmark Suite
No dependencies.
Solves the Shootout benchmarks meteor puzzle. [4]

nbody from The Python Benchmark Suite
No dependencies.
Models the orbits of the Jovian planets.

nqueens from The Python Benchmark Suite
No dependencies.
Solves the *N* 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. [5]

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.

[2]python.org/download/releases/3.4.3/
[3]kb.iu.edu/d/bezu

Benchmark	retic / python	typed / retic	typed / python
futen	1.61	1.04	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_method_slots	4.49	1.78	7.98
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

Figure 3: Performance ratios

4.2 Performance Ratios

The table in figure 3 lists three performance ratios. These ratios correspond to the extreme endpoints of gradual typing: the performance of untyped Reticulated relative to Python (the *retic/python ratio*), the performance of the fully-typed configuration relative to the untyped configuration in Reticulated (the *typed/retic ratio*), and the overall delta between fully-typed Reticulated and Python (the *typed/python ratio*).

For example, the row for `futen` reports a retic/python ratio of 1.61. This means that the average time to run the untyped configuration of the `futen` benchmark using Reticulated was 1.61 times slower than the average time of running the same code using Python 3.4.3. Similarly, the typed/retic ratio for `futen` states that the fully-typed configuration is 1.04 times slower than the untyped configuration.

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: ten are below 2x, five are between 2x and 3x, and the remaining four are below 4.5x. The typed/retic ratios are typically lower: fifteen are below 2x, one is between 2x and 3x, and the final three are below 3.5x. In particular, thirteen benchmarks have larger retic/python ratios than typed/retic ratios. This data suggests that migrating an arbitrary Python program to Reticulated adds a relatively larger overhead than migrating the same program to a fully-typed configuration.

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

4.3 Overhead Plots

Figure 4 summarizes the overhead of gradual typing in Reticulated *relative to Python* across all configurations of the nineteen benchmarks. Each overhead plot reports the percent of D-deliverable configurations (y-axis) for values of D between 1 and 10 (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 lists the benchmark's name and number of configurations. Note that the number of configurations is equal to 2^{F+C}, with F and C from figure 2.

How to Read the Overhead Plots Overhead plots are cumulative distribution functions. As the value of D increases along the x-axis, the number of D-deliverable configurations can only increase or stay the same. The important question is how many configurations are D-deliverable for low values of D. The area under the curve is the answer; more is better. A curve with a large shaded area below it implies that a large number of configurations have low performance overhead. If many benchmarks have many low-overhead configurations, a developer that applies gradual typing has a higher chance of arriving at a configuration that is D-deliverable for a value of D that meets their requirements.

After surveying the area under a curve, 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 number of D-deliverable configurations in a benchmark, the critical points are the smallest overheads a, b such that $h(a) > 0$ and $h(b) = 100$. An ideal start-value would lie between zero and one; if $a < 1$ then at least one configuration runs faster than the original Python code. The end-value b 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.

Distilling the Overhead Plots Curves in figure 4 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, seven benchmarks are 2-deliverable.

None of the configurations in the experiment run faster than the Python baseline. This is no surprise, because Reticulated adds run-time checks to Python code for each type annotation.

Eleven benchmarks have relatively smooth slopes. The plots for the other eight benchmarks have wide, flat segments

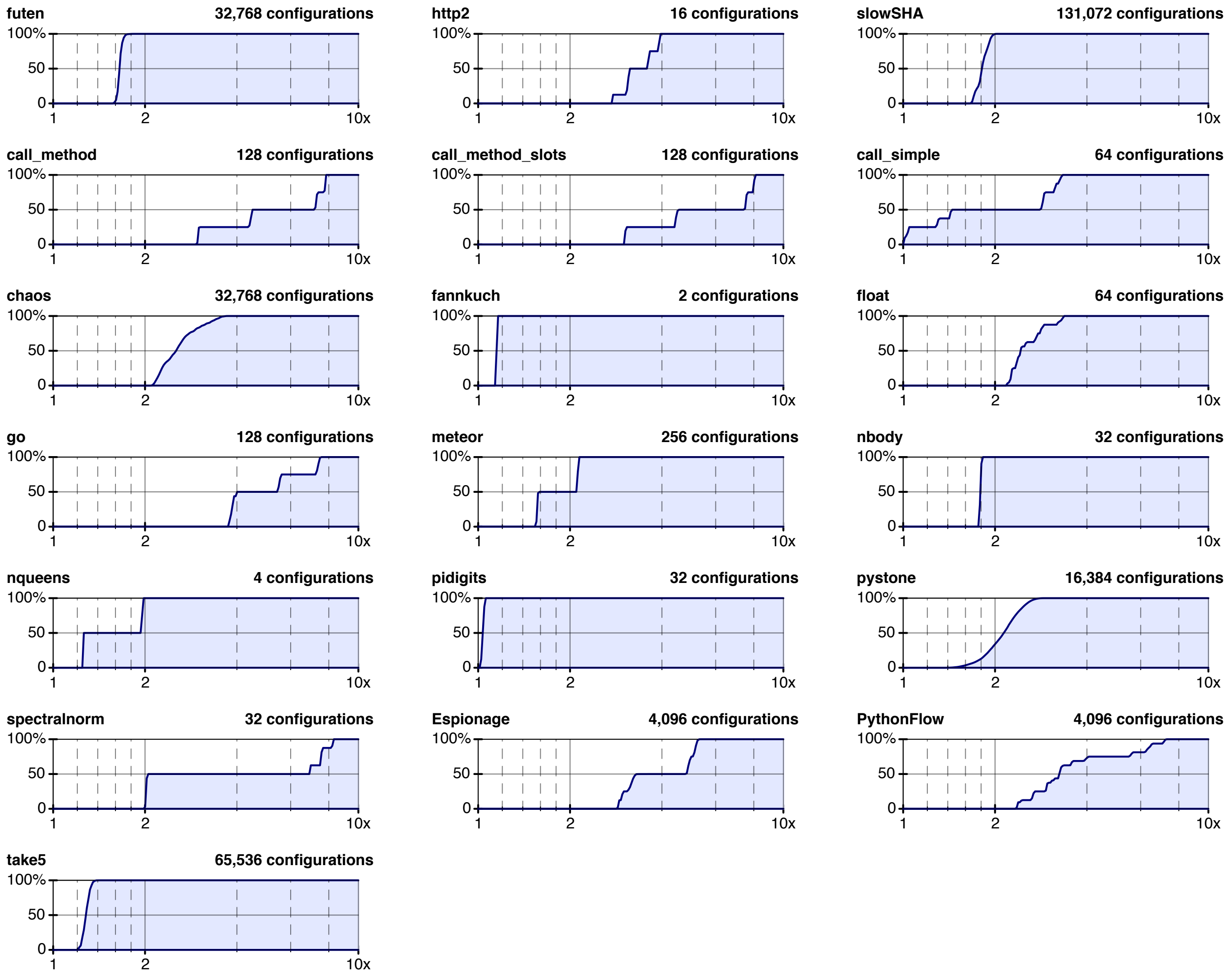

Figure 4: Overhead plots

because those benchmarks contain at least one function or method that is called frequently. For example, if a benchmark creates many instances of a class `C`, adding a type annotation to the method `C.__init__` adds significant overhead.

Sixteen benchmarks are roughly T-deliverable, where T is the typed/python ratio listed in figure 3. In these benchmarks, the fully-typed configuration is one of the slowest-running configurations. The notable exception is `spectralnorm`, in which the fully-typed configuration runs faster than 38% of the configurations. This speedup occurs because of an unsoundness in the implementation of Reticulated; in short, the implementation does not dynamically type-check the contents of tuples.[6]

4.4 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 5 demonstrate that a simple heuristic works well for these benchmarks: *the performance of a configuration is proportional to the number of typed components in the configuration.*

Figure 5 contains one green point for every run of every configuration in the experiment.[7] Each point compares the number of typed functions, methods, and classes in a configuration (x-axis) against its running time in seconds (y-axis).

The plots contain many points with both the same number of typed components and similar performance. To reduce

[6]github.com/mvitousek/reticulated/issues/36
[7]Recall from section 3.2, the data for each configuration is 40 runs.

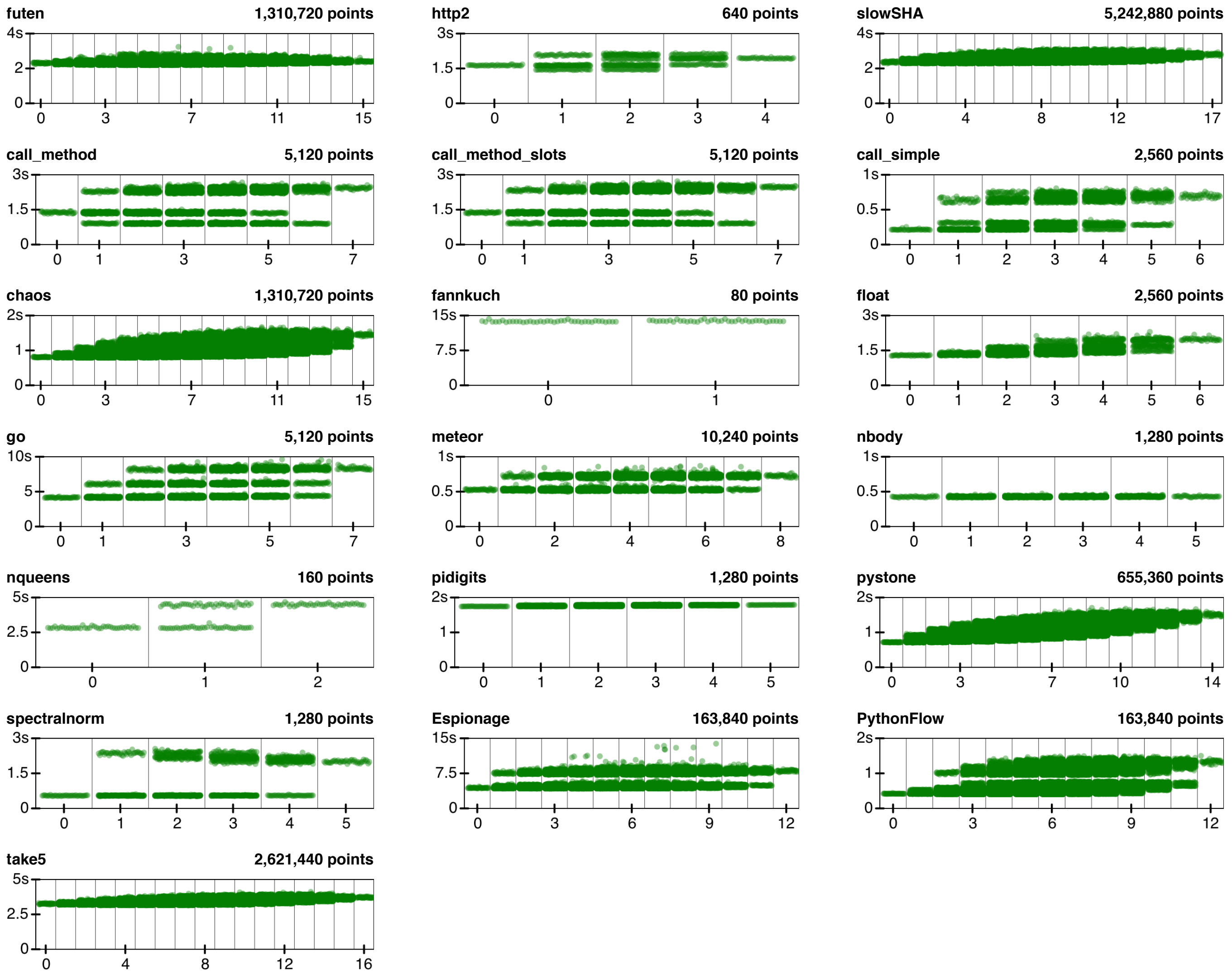

Figure 5: Running time (in seconds) vs. Number of typed components

the visual overlap between such points, the points for a given configuration are spread across the x-axis. 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: one untyped configuration with zero typed components and one fully-typed configuration with one typed component. 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.

Overall, there is a clear trend that adding type annotations adds performance overhead. The variations between individual plots fall into four overlapping categories:

Type I *(types make things slow)*: The plots for seven 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.

The `futen`, `slowSHA`, `chaos`, `float`, `pystone`, `PythonFlow`, and `take5` benchmarks are of Type I .

Type II *(types make things very slow)*: Ten 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 4.

The `call_method`, `call_method_slots`, `call_simple`, `go`, `http2`, `meteor`, `nqueens`, `spectralnorm`, `Espionage`, and `PythonFlow` benchmarks are of Type II .

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

The `fannkuch`, `nbody`, and `pidigits` benchmarks are of Type III .

Type IV *(types make things fast)*: In three benchmarks, there are some configurations that run faster than similar configurations with fewer typed components. These speedups happen for one of 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 [11]. The latter is due to a bug in the implementation (see section 4.3).

The `call_method`, `call_method_slots`, and `spectral-norm` benchmarks are of Type IV .

Note: the data for `futen`, `float`, `go`, `meteor`, and `Espionage` contain a small number of outliers. Section 6 addresses these and other threats to validity.

5 Scaling the Evaluation Method

Simple random sampling is a viable technique for approximating the number of D-deliverable configurations in a benchmark. In particular, a sampling protocol based on elementary statistics (section 5.1) is able to reproduce the overhead plots from figure 4 with a fraction of the measurements (section 5.2). Furthermore, section 5.3 demonstrates that the sampling method scales to large benchmarks. While exhaustive evaluation cannot feasibly be applied to programs containing tens of functions, the main bottleneck of the sampling technique is the cost of ascribing types to Python programs.

5.1 Sampling Protocol

Counting the proportion of D-deliverable configurations is a useful way to measure the performance overhead of gradual typing because it addresses two forms of uncertainty. First, the parameter D addresses the fact that different software applications have different performance requirements. Second, the proportion quantifies over the entire configuration space of a program—because it is impossible to predict how developers will apply gradual typing.[8] For an arbitrary configuration, the proportion of D-deliverable configurations *is* the probability that this configuration is D-deliverable.

While computing the exact proportion of D-deliverable configurations requires measuring an exponential number of configurations, a random sampling protocol can accurately and quickly approximate it. To illustrate the protocol, suppose a few developers independently apply gradual typing to a program. Each arrives at some configuration and observes some performance overhead. For a given value of D some proportion of the developers have D-deliverable configurations. There is a remote chance that this proportion coincides with the true proportion of D-deliverable configurations. Intuitively, the chance is less remote if the number of developers is large. But even for a small number of developers, if they repeat this experiment multiple times, then the average proportion of D-deliverable configurations should tend towards the true proportion. After all, if the true proportion of D-deliverable configurations is 10% then approximately 1 in 10 randomly sampled configurations is D-deliverable.

[8]The promise of gradual typing is that developers can run *any configuration*. At present, there is no data to suggest that developers are more likely to choose some configurations over others.

The following definition capture the informal sampling protocol outlined above:

Definition (95%-r, s*-approximation*) A 95% confidence interval generated from r samples, each made of s configurations is a 95%-r, s-approximation (informally, a *simple random approximation*.)

A given sample of s randomly-selected configurations contains some number n of D-deliverable configurations. The proportion n/s is the proportion of D-deliverable configurations in the sample. Repeating the sampling process r times yields a sequence of proportions; the 95% confidence interval of such a sequence is a 95%-r, s-approximation.

The theoretical justification for why this protocol should yield a useful estimate requires some basic statistics. Let d be a predicate that checks whether a given configuration from a fixed program is D-deliverable. 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 averages is normally distributed around the true proportion. Hence a 95% confidence interval generated from sample averages is likely to contain the true proportion.

5.2 Empirical Validation

In principle, a 95%-r, s-approximation should give precise and accurate bounds. It remains to be seen whether the data from a small number of samples (r) each containing a small number of configurations (s) actually do so in practice. Fortunately, the data from the exhaustive evaluation of section 4 is an adequate source-of-truth to test against.

Figure 6 demonstrates that a linear number of samples suffice to approximate the performance overhead in six benchmarks from section 4. For a benchmark containing F functions and C classes, the figure plots the confidence interval generated by ten samples of $10 * (F + C)$ configurations selected at random *with replacement*.[9] Each interval is superimposed on the matching overhead plot from figure 4.

These particular 95%-10, $10(F+C)$-approximations all contain the the true number of D-deliverable configurations for

[9]The theoretical justification in section 5.1 assumes random sampling without replacement, but the chance of drawing the same configuration twice is quite small, and removing this chance slightly increases the odds of drawing an extreme outlier.

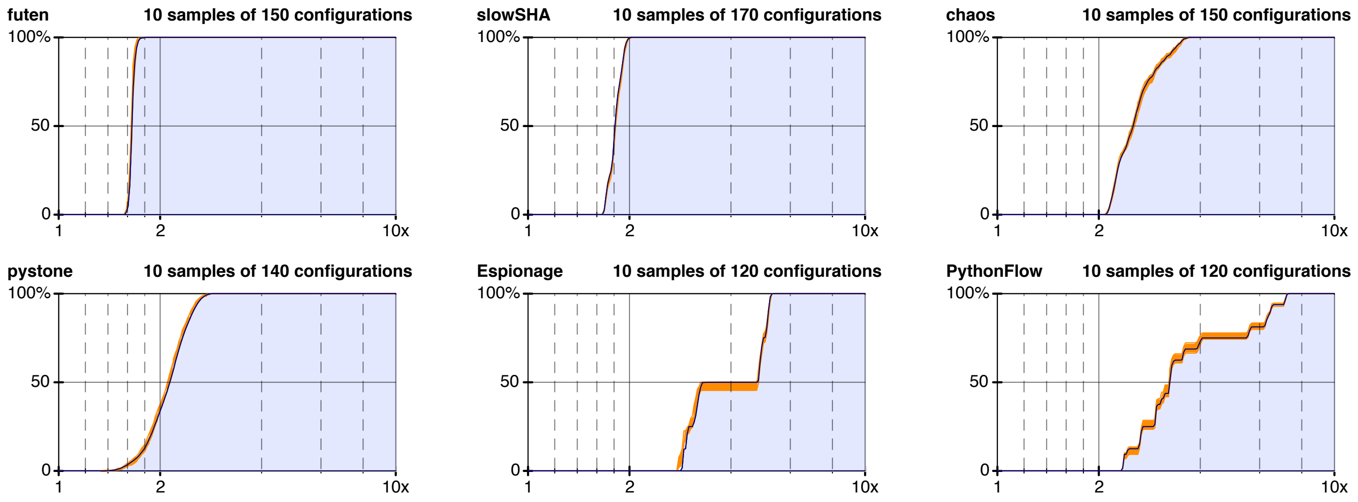

Figure 6: Validating the simple random approximation method

values of D between 1 and 10. The intervals are futhermore small, and thus practical substitutes for the overhead plots.

5.3 Approximate Evaluation

This section presents the results of an approximate performance evaluation of three benchmark programs with large configuration spaces:

Benchmark	SLOC	M	F	C
`sample_fsm`	148	5	17	2
`aespython`	403	6	29	5
`stats`	1118	13	79	0

Two of these programs, `aespython` and `stats`, originate from case studies by Vitousek et al. [11]. The following descriptions provide more information about each benchmark's size and purpose.

sample_fsm from Linh Chi Nguyen
Depends on the `itertools`, `os`, and `random` libraries.
Simulates the interactions of economic agents via finite-state automata [5]. This benchmark is adapted from a similar Racket program called `fsmoo` [4].

aespython from Adam Newman and Demur Remud
Depends on the `os` and `struct` libraries.
Implements the Advanced Encryption Standard. Uses the `os` library only to generate random bytes and invoke the `stat()` system call.

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.

5.3.1 Results

Figure 7 plots the results of applying the protocol in section 3.2 to random configurations. Specifically, the data for a benchmark with F functions and C classes consists of ten samples of $10(F+C)$ configurations selected without replacement. These results confirm many trends from section 4.3:

- No configurations run faster than the Python program. The lowest overheads range between 1.1x and 4x.
- All configurations are 10-deliverable.
- Most configurations are T-deliverable, where T is the benchmark's typed/python ratio (marked on each plot's x-axis).
- The curves have smooth slopes, implying the cost of annotating a single function or class is low.
- The intervals are tight.

6 Threats to Validity

Our work may suffer from two problems: measurement error and systematic bias. This section spells out the details.

6.1 Measurement Error

The data are timings recorded on the Karst at Indiana University cluster using the Python function `time.process_time()`. Assuming `process_time()` is accurate, the cluster infrastructure is prone to at least two sources of error. First, cluster nodes may have non-uniform performance despite being identical servers. Second, the load on other nodes in the cluster may affect the latency of system calls. These measurement biases may explain the outliers evident in figure 5.

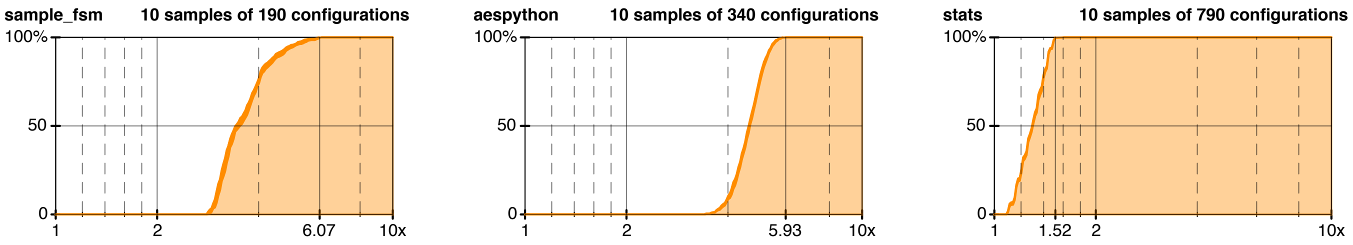

Figure 7: Simple random approximation plots

6.2 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[10] reveals that even small packages have many 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, five benchmarks are either missing annotations or use the dynamic type. The `futen` benchmark contains a class with an untyped field and the `take5` benchmark contains a function that accepts an optional argument.[11] These are mistakes on our part, but random sampling with and without the errors yields similar results. Three benchmarks (`go`, `pystone`, and `stats`) use the dynamic type to overcome limitations in Reticulated's 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 and classes may come with entirely different performance. We leave this as an open question.

7 Why is Reticulated so Fast ...

The worst observed slowdown of Reticulated over Python is within one order of magnitude. By contrast, many partially typed Typed Racket programs are two orders of magnitude slower than Racket [4, 8]. While implementation technology and the peculiarities of the programs affect performance, this order-of-magnitude gap suggests fundamental differences.

We have identified three factors that reduce the overhead of Reticulated relative to Python. First, Reticulated's type system lacks support for common Python idioms. Second, Reticulated's error messages rarely provide actionable feedback. Third, Reticulated guarantees an alternative notion of type soundness.

7.1 Missing Types

Reticulated currently lacks union types, recursive types, and types for variable-arity functions. Consequently, Reticulated cannot fully-type some programs in our experiment. One common issue is Python code that uses `None` as a default value. The benchmark versions of such code use well-typed defaults instead. Other benchmark versions resort to dynamic typing. Both `pystone` and `stats` suffer from the lack of union types, and go contains a recursive class type. Lastly, we tried typing a Lisp interpreter, but the program made too-heavy use of union and recursive types.

Rewrites are time-consuming and prone to introduce bugs; mandatory dynamic typing contradicts the goals of gradual typing. Thus, it would benefit developers if Reticulated followed *PEP 484*: *Type Hints* and added support for unions, recursive types, and functions with optional, variable, and keyword arguments.

Enforcing these types at run-time, however, will impose a higher cost than the single-test types that Reticulated programmers must currently use. A union type or (equi-)recursive type requires a disjunction of type tests, and a variable-arity procedure requires a sequence of type checks. If, for example, every type annotation τ in our benchmarks were a union type with τ and `Void`, then overall performance would be nearly twice as worse as it currently is.

7.2 Uninformative Errors

Errors matter [3]. When systems work, everyone is happy, but when systems break, developers want error messages that pinpoint the source of the fault.

Two kinds of faults can occur in Reticulated: static type errors and dynamic type errors. A static type error is a mismatch between two types. A dynamic type error is the result of a mismatch between a type annotation and an untyped value. Typically, a dynamic type error occurs long after the value enters typed code.

When Reticulated discovers a static type error, it reports the current line number and the conflicting types. To its credit, this information often pinpoints the source of the fault.

[10]pypi-ranking.info/alltime
[11]Reticulated currently ignores the type signatures of functions with optional arguments, see github.com/mvitousek/reticulated/issues/32.

When Reticulated discovers a dynamic type error, it prints a value, the name of the check that failed, and a stack trace. This information does little to help diagnose the problem. For one, the relevant type annotation is not reported. A programmer must scan the stack trace for line numbers and consider the type annotations that are in scope. Second, the value in the error message may not be the value that caused the error. For instance, the reported value may be an element of an ill-typed data structure or a return value of an ill-typed function. Third, the relevant boundary is rarely on the stack trace when the program raises the check error. The stack may contain only well-typed code (see the appendix for an example).

Refining the dynamic error messages will add performance overhead. For example, Vitousek et al. [12] built an extension to Reticulated that reports a set of potentially-guilty casts when a dynamic type error occurs. They report that tracking these casts can double a program's typed/python ratio.

7.3 Alternative Soundness

Sound type systems are useful because they provide guarantees. A sound *static* type system guarantees that evaluating a well-typed program can result in one of three possible outcomes: evaluation terminates with a value of the same type; evaluation diverges; or evaluation raises an error from a well-defined set. A sound *gradual* type system cannot provide the same guarantee because it admits untyped code. Thus, a gradual type system must redefine soundness.

One approach is to *generalize* traditional type soundness with a fourth clause to cover interactions between typed and untyped code. Typed Racket takes this approach [10]. In particular, if a program e is well typed at type τ, then either:

1. e reduces to a value v with type τ;
2. e diverges;
3. e signals an error due to a partial primitive operation; or
4. e raises an exception that points to the guilty boundary [2] between typed and untyped code.

A second approach is to *modify* soundness. Reticulated [12] takes this approach, and weakens the first and fourth clauses:

$1'$. e reduces to a value v with type tag $\lfloor\tau\rfloor$;
$4'$. e signals an exception that points to a set of potentially guilty boundaries between typed and untyped code.

The type tag of τ is its outermost constructor. For example, the type tag of `List(Int)` is `List`. The set of boundaries is always empty in the version of Reticulated that was public when we began our evaluation [11]; section 7.2 addresses the performance implications of $4'$.

As figure 8 demonstrates, the modified clause $1'$ implies that a Reticulated term with type `List(String)` may evaluate to a list containing any kind of data. On one hand, this fact is harmless because type-tag soundness implies that any read from a variable with type `List(String)` in typed code is tag-checked. On the other hand, Reticulated does not monitor values that leave a typed region. Thus, two interesting scenarios can arise:

The *typhoid mary* scenario Typed code can create an ill-typed value, pass it to untyped code, and trigger an error by violating an implicit assumption in the untyped code. The source of such "disguised" type errors may be impossible to pinpoint.

The *sybil* scenario Two typed contexts can safely reference the same value at incompatible types.

It remains to be seen whether these potential scenarios cause serious issues in practice. Developers may embrace the flexibility of alternative soundness and use Reticulated in combination with unit tests.

The performance implications of $1'$ are substantial. A gradual type system that enforces traditional soundness must exhaustively traverse data structures before they leave typed code, and must monitor functional values to ensure their future applications are well-typed. Enforcing $1'$ requires a tag check, nothing more.

```
def make_strings()->List(String):
  xs = []
  for i in range(3):
    if   i == 0: xs.append(i)
    elif i == 1: xs.append([True])
    else       : xs.append(make_strings)
  return xs

make_strings()
```

Figure 8: A well-typed Reticulated program

8 Is Sound Gradual Typing Alive?

The application of Takikawa et al.'s method to Reticulated appears to indicate that at least one sound, gradually typed language comes with a performant implementation. In particular, the overhead plots for Reticulated look an order of magnitude better than those for Typed Racket. Appearances are deceiving, however. Reticulated's type system is far less expressive than Typed Racket's. Furthermore, its error messages, especially for higher-order values, are seldom actionable. Most importantly, though, Reticulated guarantees an alternative soundness. A program of type `List(String)` may evaluate to a list of integers—without signaling any violation.

Our evaluation effort thus confirms a widely held conjecture and leaves us with an open research problem. While the sacrifice of traditional soundness improves the performance of gradual typing systems, it remains unclear whether programmers will accept what remains if they want to reason with types.

Bibliography

[1] Kenneth R. Anderson and Duane Rettig. Performing Lisp analysis of the FANNKUCH benchmark. *ACM SIGPLAN Lisp Pointers* 7(4), pp. 2–12, 1994.

[2] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Symposium on Programming*, pp. 214–233, 2012.

[3] Matthias Felleisen. From POPL to the classroom and back. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 126–127, 2002.

[4] 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.

[5] Linh Chi Nguyen. Tough Behavior in Repeated Bargaining game, A Computer Simulation Study. Master in Economics dissertation, University of Trento, 2014.

[6] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Scheme and Functional Programming Workshop*, 2006.

[7] 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.

[8] 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.

[9] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. Dynamic Languages Symposium*, pp. 964–974, 2006.

[10] 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.

[11] 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.

[12] 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.

Appendix

Figure 9 is a small Reticulated program that signals a type error at run-time. The error message is reproduced in figure 10. It consists of a stack trace and the name of a failing check, as described in section 7.2. Notice, in particular, how the stack frames for `myfile.py` guide developers to typed code.

```
def make_strings()->List(String):
    xs = []
    for i in range(3):
        if   i == 0: xs.append(i)
        elif i == 1: xs.append(True)
        else       : xs.append(make_strings)
    return xs

def get_lengths(los:List(String))->List(Int):
    return [strlen(s) for s in los]

def strlen(s:String)->Int:
    return len(s)

strs = make_strings()
get_lengths(strs)
```

Figure 9: Reticulated program `myfile.py`

```
Traceback (most recent call last):
  File "/.../retic", line 9, in <module>
    load_entry_point('retic==0.1.0',
                     'console_scripts',
                     'retic')()
  File "/.../retic/retic.py", line 155, in main
    reticulate(program,
               prog_args=args.args.split(),
               flag_sets=args)
  File /.../retic/retic.py", line 104, in reticulate
    utils.handle_runtime_error(exit=True)
  File "/.../retic/retic.py", line 102, in reticulate
    _exec(code, __main__.__dict__)
  File "/.../retic/exec3/__init__.py", line 2, in _exec
    exec (obj, globs, locs)
  File "myfile.py", line 16, in <module>
    get_lengths(strs)
  File "myfile.py", line 10, in get_lengths
    return [strlen(s) for s in los]
  File "myfile.py", line 10, in <listcomp>
    return [strlen(s) for s in los]
  File "myfile.py", line 12, in strlen
    def strlen(s:String)->Int:
  File "/.../retic/runtime.py", line 109, in
      check_type_string
    return val if isinstance(val, str) else rse()
  File "/.../retic/runtime.py", line 88, in rse
    raise Exception(x)
Exception: None
```

Figure 10: Reticulated's error message for figure 9