Which of my Transient Type Checks are not (Almost) Free?

Isaac Oscar Gariano
Engineering and Computer Science
Victoria University of Wellington
New Zealand
Isaac@ecs.vuw.ac.nz

Richard Roberts
Computational Media Innovation Centre
Victoria University of Wellington
New Zealand
rykardo.r@gmail.com

Stefan Marr
School of Computing
University of Kent
United Kingdom
s.marr@kent.ac.uk

Michael Homer
Engineering and Computer Science
Victoria University of Wellington
New Zealand
mwh@ecs.vuw.ac.nz

James Noble
Engineering and Computer Science
Victoria University of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

Abstract

One form of type checking used in gradually typed language is *transient type checking*: whenever an object 'flows' through code with a type annotation, the object is dynamically checked to ensure it has the methods required by the annotation.

Although naïve implementations of transient type checks have a high runtime overhead, just-in-time compilation and optimisation in virtual machines can eliminate much of this overhead. Unfortunately the improvement is not uniform: while most type checks can be optimised away, some will significantly decrease a program's performance, and some may even increase it.

In this paper, we refine the so called "Takikawa" protocol, and use it to identify which type annotations have the greatest effects on performance. In particular, we show how graphing the performance of such benchmarks when varying which type annotations are present in the source code can be used to discern potential patterns in performance. We demonstrate our approach by testing the Moth virtual machine: for many of the benchmarks where Moth's transient type checking impacts performance, we have been able to identify one or two specific type annotations that are the likely cause. Without these type annotations, the performance impact of transient type checking becomes negligible.

Using our technique programmers can optimise programs by removing expensive type checks, and VM engineers can identify new opportunities for compiler optimisation.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; *Object oriented languages*; *Interpreters*;

Keywords dynamic type checking, gradual types, optional types, Grace, object-oriented programming

VMIL'19, October 22, 2019, Athens, Greece
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Which of my Transient Type Checks are not (Almost) Free?. In *Proceedings of ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'19).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Gradual typing aims to add static type annotations to dynamic languages, increasing their safety while maintaining flexibility [12, 36, 38] — or, complementarily, to permit dynamic type annotations within static languages, increasing flexibility whilst maintaining safety [1].

There is a spectrum of different approaches to gradual typing [15, 19]. At one end is "pluggable types" (as in Strongtalk [13]) or "erasure semantics" (as in TypeScript [8]), where all types are erased before execution, limiting the benefit of types to statically typed portions of programs, while not providing any for of run time checking. In the middle there are "transient" or "type-tag" checks (as in Reticulated Python), which offer first-order semantics, by checking that an object's type constructor or supported methods match any static types that the object flows through. [11, 20, 33, 37, 44]. Reticulated Python also supports alternative "monotonic" semantics which mutate objects by narrowing their concrete types when passed into contexts with more specific types. On the other end of the spectrum are behavioural type checks (as in Typed Racket [42, 43], Gradualtalk [3]), and "proxies" (provided by Reticulated Python), which support higher-order semantics, retaining types until run time, and performing checks eagerly, thus detailed information about type violations can be given as soon as possible via "blame" tracking [2, 47]. Finally, there is also ductile typing, which dynamically interprets a static type system at run time [6]. Unfortunately, most gradual systems with run-time semantics (as opposed to type erasure as in TypeScript) typically imposes significant run-time performance overheads to provide these semantics. This has lead to a significant body of research, on one hand developing techniques to optimise gradual typing [5, 20, 30, 32, 46] and on the other, developing techniques to evaluate the resulting performance [21, 41].

This paper builds upon our recent work on optimising transient type checks [34, 35] by investigating how benchmark results can be repurposed to identify which particular transient type checks are resistant to optimisation.

Specifically, the contributions of this work are:

- an approach to identifying gradual type annotations that cause significant performance effects
- an observation that the overhead of Moth's transient type checking on small benchmarks (with 10–250 type annotations) is usually caused by only one or two type annotations

The next section discusses dynamic type checks and gradual typing in Moth (an implementation of the Grace language), then section **??** describes our benchmarking protocol. Section 4 then presents the overall results of our benchmarks, while section 5 looks at the results of benchmarking individual type checks. Section 6 presents some additional related work, and finally 7 summarises our results, and briefly considers threats to validity.

2 Background

Our work is based on the Moth virtual machine [34, 35], an implementation of the Grace programming language [9, 14]. Moth is based on the Graal and Truffle toolchain [49, 50], and developed from a Newspeak implementation based on the Simple Object Machine [17, 28].

2.1 Grace and Transient Type Checking

Grace is an object-oriented, imperative, educational programming language, with a focus on introductory programming courses, but also intended for more advanced study and research [9, 14]. While Grace's syntax draws from the so-called "curly bracket" traditions of C, Java, and JavaScript, the structure of the language is in many ways closer to Smalltalk: all computation is done via dynamically dispatched "method requests" where the object receiving the request decides what code to run, control structures are built out of lambda expressions support "non-local" returns, i.e. they can return to the point where execution first encountered the lambda [18]. In other ways, Grace is closer to JavaScript than Smalltalk: Grace objects are created from object literals, rather than by instantiating classes [10, 25] and objects and classes can be deeply nested within each other [26].

Grace's Typing In Grace, all declarations can be annotated with types. As Grace is designed to support a variety of teaching methods, implementation of Grace are free to check such type annotations statically, dynamically, or not at all. The type system of Grace is intrinsically gradual: type annotations should not affect the semantics of a correct program [38]. The type system includes a distinguished "Unknown" type which matches any other type; this unknown type is the default when type annotations are omitted.

Static typing for the core of Grace's type system has been described elsewhere [24]; here we explain how these types can be understood dynamically, from the Grace programmer's point of view. Grace's types are structural [9], that is, an object conforms to a type whenever it conforms to the "structural" requirements of a type, rather than requiring classes or objects to explicitly declare their intended type.

In Grace, types specify a set of method signatures that an object must provide. A type expresses the requests an object can respond to, for example whether a particular accessor is available, rather than a location in a class hierarchy.

Moth's Transient Type Checking Moth's implementation of transient type checks are only only first-order. Moth only checks dynamically that an object has methods of the same name and arity as are required by a type: any argument and return types of such methods are not checked.

In particular, Moth performs the following type checks at run time:

- when a method is requested, arguments that are passed are checked against the corresponding parameter type annotations of the called method, this is done before the body of the method is executed;
- when the body of a method has finished executing, but before it returns to its caller, the method's return value is checked against the return type annotation of the called method;
- whenever a variable is read or written to, its value is checked against the type specified by the variables declaration.

To see how this works in practice, consider this piece of Grace code:

```
1 def o = object {
2    method three -> Number {3}
3 }
4 type ThreeString = interface {
5    three -> String
6 }
7 def t : ThreeString = o
8 printNumber (t.three)
```

Moth will perform dynamic type checks:

- on line 7, when the `o` object initialises the variable `t`, Moth checks that `o` has a 0-argument method called `three`;
- on line 8, when the value of `t` is read, Moth checks that its value (`o`) still has a `three` method;

- on line 2, when the method requested by "`t.three`" returns, Moth checks that returned value conforms to the `Number` type; and (presumably) within the definition of `printNumber(n : Number)` (not shown), Moth will again check that the value is a `Number`.

Note that we never check whether the result of requesting "`t.three`" is actually a `String` (as one may expect from line 5) because Moth only performs first-order type checks (it checks whether objects have conforming methods) not higher-order checks (whether the argument and result types of methods' conform). In addition, Moth only checks when values flow through explicit type annotations. This is why the type declared in lines 4-6 is checked only on line 7 (where it is mentioned explicitly); and the check only requires the presence of a method called `three`, regardless of the method's declared return type.

Moth's Optimisation We are developing Moth as a research platform [35]. Like other VMs based on the Truffle and Graal toolchain, Moth is a self-optimising AST interpreter [51]. The key idea is that an AST rewrites itself based on a program's run time values to reflect the minimal set of operations needed to execute the program correctly. The rewritten AST is then compiled into efficient machine code. This rewriting often depends on the dynamic types of the objects involved. In the simplest case, a "self" call (when one method on an object requests a second method on the exact same object) will always result in executing the exact same method. Thus the called method can be inlined into the callee, avoiding overhead of a machine-level subroutine invocation and an object-oriented dynamic dispatch.

Moth relies on a number of standard techniques for optimising object-oriented programs. "Shapes" [48] capture information about objects' structures and (run time) field types, allowing a just-in-time compiler to represent objects in memory similarly to C structs and, consequently, can generate highly efficient code. "Polymorphic inline caches" [22] use object shapes to cache the results of method lookups, avoiding expensive class hierarchy searches or indirect jumps through virtual method tables. Since Moth is built on the Truffle framework, Graal comes with additional support for partial evaluation, which enables efficient native code generation for Truffle interpreters [49].

3 Experimental Methodology

Our goal is to identify which type annotations in Grace programs cause performance effects. To this end, we built upon the so-called "Takikawa" or "Takikawa-Greenman" evaluation protocol [21, 41]. It uses 2^N *configurations* of each benchmark. A configuration is a particular mix of static and dynamically typed code, forming a lattice of configurations. We only test a relatively small sample of this lattice, which is in our experience sufficient to pinpoint performance anomalies caused by type annotations.

3.1 The Takikawa Protocol

The Takikawa evaluation protocol was originally proposed for Typed Racket, where static vs dynamic typing is set per-module, so N is the number of modules. Grace, following other languages such as Reticulated Python [39, 44, 46], allows programmers to choose whether each individual declaration should be type-checked. This means N in Grace is the number of type annotations in the program, and so checking an entire lattice for even a moderately sized benchmark would be infeasible. Vitousek et al. modified the Takikawa protocol for these kinds of languages by taking an approach based on sampling [45]. The Takikawa-Vitousek protocol divides the number of type annotations in a fully-typed program into a maximum of 100 intervals, and then randomly generates ten programs within each interval by erasing type annotations.

To identify type annotations causing anomalies, we adapted the Takikawa protocol and took inspiration from the Takikawa-Vitousek variant. For each benchmark, we generated 100 partially typed versions, or fewer if the benchmark has less than 11 types. We did an even split so that for each interval $i \geq 1$ and $i < N$, we generated roughly the same number of configurations with i type annotations. We used Robert Floyd's sampling algorithm [7] to choose the type annotations each configuration contained randomly, and we ensured that no duplicate configurations were generated. In addition to these, we tested fully untyped and typed versions, for a total of 102 configurations per benchmark (or 97 in the case of our Storage benchmark, since it only has 10 type annotations).

3.2 The Benchmarks

For this work, we rely on the benchmark suite compiled for previous work [35]. It is a collection of 21 benchmarks in total, derived from the Are We Fast Yet benchmark suite [29] and other benchmarks from the gradual-typing literature.

Each benchmark has a complete set of type annotations, i.e., all possible type annotations are present, which is verified in the parser.

In our previous work, we [35] we determined that the overhead of type checking on Moth is on average of 5% (min. -13%, max. 79%). This compares the peak performance of Moth with all checks disabled against an execution that has all checks enabled.

3.3 Experimental Set Up

As in our previous work [35] which used the same set of benchmarks, to account for the complex warmup behaviour of modern systems [4] as well as the non-determinism caused by e.g. garbage collection and cache effects, we ran each benchmark for 1000 iterations in the same invocation of Moth, and discard the first 350 iterations to ignore warmup JIT compilation.

Though outliers remain visible in the plots for each individual benchmark, the largest 95% confidence interval we obtained (over the mean time after warmup) for any of experiments was ±3.36% (for the PyStone benchmark).

All experiments were executed on a Windows 10 machine running Fedora 30 within Windows Subsystems for Linux (version 1). The machine has an Intel Core i7-6800K @3.40GHz, with 6 cores, for a total of 12 hyperthreads. This computer appears to produce comparable results to the one used in our previous work, although the absolute performance is higher, the relative overhead of transient typechecks is similar.

We used the same version of Moth as previously, together with ReBench 1.0 [27], Java 1.8.0_191 Graal 0.43. Benchmarks were executed one by one to avoid interference between them. The analysis of the results and plots where generated using Python 3.7.3 and PGFPLOTS 1.16. To enable reproductions, the scripts we used to generate and run our experiments, including the source code for all the configurations tested, are available online.[1]

4 Performance of Benchmark Configurations

Before we start to investigate specific type annotations, we present the performance measurements of our sample of the typing lattice configurations in figure 1. These results are the foundation for a more detailed analysis.

Following [45], the points on each graph in figure 1 show the average execution of each individual configuration. The x-axis represents the proportion of type annotations for each configuration, with the left- and right-most points showing the times for the fully untyped and typed configurations respectively. The execution time in milliseconds is shown on the left y-axes, and time relative to the fully untyped configuration is shown on the right y-axes.

Most of the graphs are essentially horizontal lines, indicating that the overhead of including type annotations is negligible. The plots for CD and Richards show a roughly linear increase, however, i.e. for these two benchmarks, adding type annotations reduces performance linearly. On the other hand, the plots for Go, Permute, DeltaBlue, and Storage show *decreases*: i.e. adding more type annotations *improves* performance of these benchmarks.

By inspecting the scatterplots, we observe that the performance of Moth in almost any configuration of a benchmark is bounded by the performance in the untyped and fully-typed configuration. That is, for these benchmarks on transient type checks, measuring just the untyped and fully-typed configurations would provide excellent estimates of a benchmark's performance bounds. This is different to the experience of other kinds of gradual typing, where the best, and most importantly worst, configurations are not always those fully typed or fully untyped [21]. However, the Richards and Snake benchmarks *do* have sections outside these bounds, and isolated executions of a few others (SpectralNorm, JSON, DeltaBlue) are also outliers. We believe that the 0% and 100% bounds are nonetheless a reasonable heuristic estimator in most cases, and will examine further the outlying benchmarks.

Of particular note is that some of the graphs (Permute, Storage, Snake, Towers, and Richards) appear to show bimodal performance profiles, that is, two separate roughly-horizontal lines. Presumably Moth can remove all the overhead from some configurations, but in others there must be one or more type checks that cannot be optimised away. List appears to have three performance modes: most of the configurations up to about 60% annotations perform at roughly the same speed as untyped code; most configurations above 70% perform 1.8 times more slowly than untyped code; and then across all annotation percentages there are configurations that perform roughly 1.5 times slower than the untyped baseline.

5 Identifying Type Annotations With Signification Performance Impact

We hypothesise that the previously identified bi/tri-modal performance behaviour as seen in the graphs for Permute and others (cf. figure 1) are caused by only a few type annotations: i.e. there are a very few annotations that determine each benchmark's performance.

To verify this hypothesis for each type annotation we measured one additional configuration, with only that single type annotation present. We did this to compare the overhead of each type annotation in isolation against the no-typecheck baseline.

Figure 2 shows the results of these experiments. It shows a pair of graphs for a selection of ten benchmarks: it's associated typing lattice scatter plot and a column graph showing the results of these single type annotation experiments.

The column graphs, right of the corresponding scatter plot, shows the execution time of configurations with only a single type annotation, the x-axis indicates the index of this annotation, thus the first column represents the first annotation, and the last column represents the last one (in the order they appear in the source code). The y-axes for the column graphs are the same as the associated scatter graphs. However, though the right y-axes for the column graphs appear to be identical to the scatter plot ones, they are relative to performance results collected on different days: this was to ensure that the performance of the test computer had not changed significantly from when the typing lattice experiments were run.

For each benchmark we highlighted one or two type annotations that appear to show a pattern for the typing lattice performance scatter plots, or in the case of SpectralNorm

[1] https://gitlab.ecs.vuw.ac.nz/isaac/Moth-Takikawa

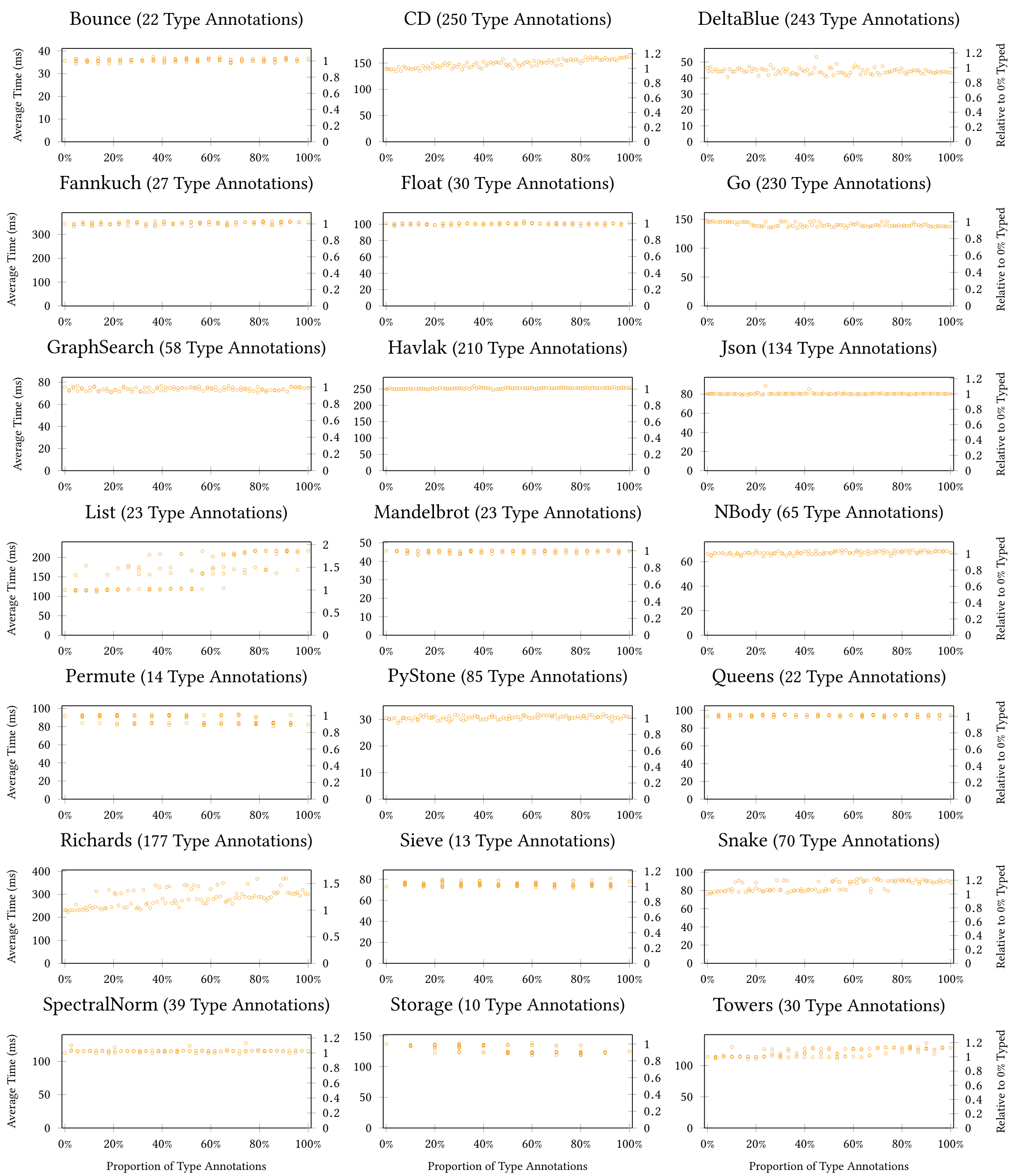

Figure 1. Graphs of (at most) 102 configurations in the typing lattices for each benchmark. Time is measured as the mean of the 351st to the 1,000th benchmark iteration under a single invocation of Moth (lower is better).

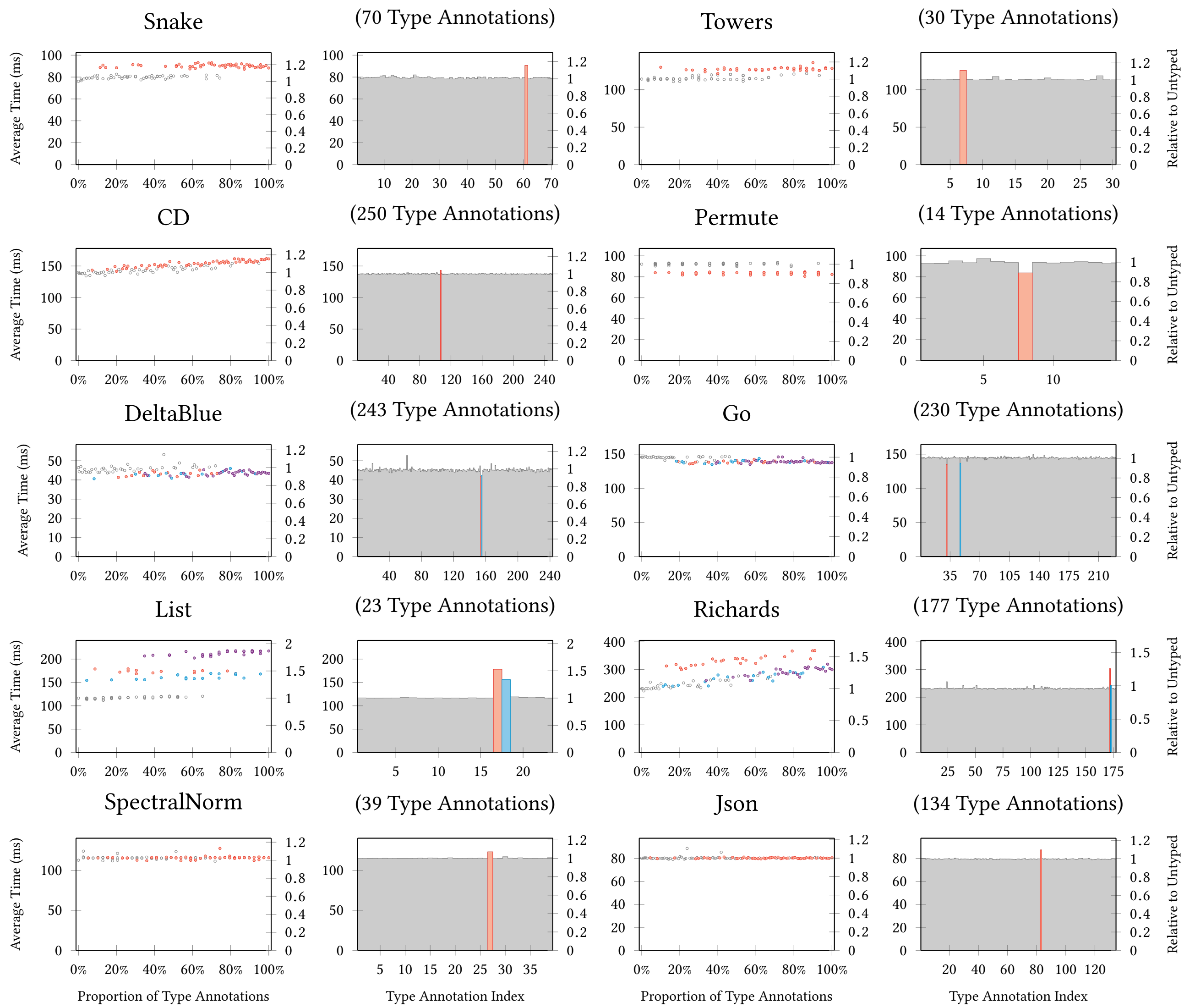

Figure 2. Pairs of colour coded scatter and column graphs. The scatter graphs represent the performance of a sample of the typing lattices. The column graphs show the performance of every configuration with only one type annotation. The scatter plots and column graphs are colour coded based on whether a particular type annotation or two are present in the source code.

and Json had higher than usual columns. The identified types are those represented by the red and blue columns. The scatter plots are colour-coded accordingly: a red or blue circle represents configurations with the given type annotation present, but not both, purple circles represent configurations with both type annotations present, and grey circles represent those with neither. Though we exhaustively inspected such colour coded scatter plots for all 1,775 type annotations across all 21 benchmarks, the only patterns we noticed are those shown in 2.

As can be seen, most of the time the patterns we found in the typing lattice graphs correspond to outliers in the single type annotation column graphs. For the Snake, Towers and CD benchmarks, there is clearly a pattern where an individual type annotation (highlighted in red) appears to be the cause of the slowest/upper half of the typing lattice graphs. For Permute there is actually a significant performance *increase*, indicated by the lower half of the lattice being highlighted in red. DeltaBlue and Go also show a slight increase in performance, but here this is caused by two type annotations (in red and blue); this effect does not appear to be cumulative, however: the purple dots are about the same height as the red and blue ones, so *either* annotation is sufficient for the full benefit.

List is interesting as it demonstrates that the red and blue type annotations both cause a significant decrease in performance, which is even greater when both are present. Richards is particularly odd as it shows a performance decrease that appears to occur when the red type annotation is present, but not the blue one. We had previously identified this benchmark and the aforementioned type annotations [35] as being the worst case for Moth.

For SpectralNorm and Json, although we found outliers in the column graphs (types 27 and 83, highlighted in red), we were unable to find a matching pattern in the configuration performance. Finally, for DeltaBlue and Richards, there are two noticeable spikes in the column graphs (at #62 and #24, respectively), however we again found no apparent relation between these outliers and the overall performance

In particular, observe that the grey dots for Snake, Towers, Permute, DeltaBlue, Go, and List are mostly flat horizontal lines, this indicates that by simply deleting the red and blue type annotations, the performance impact of transient type checking becomes negligible. However for CD and Richards, the transient type checking overhead of the grey dots appears linear, albeit still less than with the red and blue type annotations present. Thus we can observe that usually, only a couple of type annotations are responsible for the overhead caused by Moth's transient typed checking, whereas the rest are "free".

The remaining benchmarks, which we have not shown in figure 2, have relatively flat column graphs and we could not identify any patterns in the typing latticse. The benchmark with the greatest difference in single-type configurations relative to the untyped configuration is Storage, with a configuration 7.9% faster than the untyped baseline. However, though Storage (as shown in 1) appears to be bimodal like Towers, we were unable to find any relationships between its typing lattice and any individual type annotations.

6 Related Work

The high-performance computing community has been investigating how tools and visualisations can help developers to utilise their systems more efficiently [16, 31]. Their focus is typically on parallelisation opportunities, guided by runtime feedback, cost models, or heuristics. Their large body of work [23] uses various approaches, though, we are not aware of work that has used an approach similar to ours.

At the moment, our approach to identifing type annotations that case performance anomalies is not integrated into a development environment. Though, for instance Optimization Coaching [40] is a promising direction. Optimization Coaching uses feedback from the runtime to guide developers to insert or change type declarations to enable a compiler to generate a more optimal program. In this spirit, we would eventually want to achieve the same, although in our case, we need to run full experiments to get the necessary information.

7 Discussion and Conclusion

In this paper we have investigated how benchmarks can be repurposed to determine precisely which transient type checks are resistant to the optimisations provided by a just-in-time compiling virtual machine. We observed that many of our benchmark results conducted under the Takikawa protocol seemed to have a characteristic bimodal performance profile: some of the benchmark configurations ran significantly slower than the remaining configurations. We also observed trimodal profiles, as well as performance increases when type annotations where added.

By inspecting graphs of the performance of where only one type annotation is present, we can easily identify type annotations that likely have a significant effect on performance. By then inspecting the typing lattice graphs colour coded based on whether such type annotation was present or not, we were often able to notice patterns. In particular, these patterns suggested the just one or two type annotations are likely responsible for the bimodal performance and most of the overhead caused by Moth's transient type checking. Though every type annotation that appeared to have a significant effect across the typing lattice also showed a significant performance effect when no other typing annotations where present, the converse did not hold.

This is preliminary work, in particular we have not yet identified exactly why these type annotations appear to have an effect on performance. There are also a number of threats to validity. Regarding construct validity, our underlying implementation may contain undetected bugs that affect the semantics or performance of the gradual typing checks. Regarding internal validity, our benchmarking harness runs on the same implementation and therefore is subject to the same issues. Regarding external validity, Moth is built on the Truffle and Graal toolchain, so we expect to resemble other Graal VMs doing similar AST-based optimizations of transient type checks. Because we rely on common techniques, we expect our results to be transferable to other JIT implementations as well.

Finally, it is not clear how our results would transfer to other gradually typed-languages or other semantics for gradual typing. Our benchmarks do not depend on any features of Grace that are not common in other object-oriented languages, but as Grace lacks a large corpus of programs the benchmarks are necessarily small and artificial. The advantage of Grace for this research is that their relative simplicity means we have been able to build an implementation that features competitive performance with significantly less effort than would be required for larger and more complex languages.

In the future, we hope to investigate statistical techniques to determine the significance of each type annotation's contribution to a programs overall performance. We would also like to investigate whether this approach can assist with optimsations for programmers' day-to-day development, or help VM engineers identifying performance bugs in the underlying virtual machines.

References

[1] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 237–268.

[2] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 201–214.

[3] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2014. Gradual typing for Smalltalk. *Sci. Comput. Program.* 96 (2014), 52–69. https://doi.org/10.1016/j.scico.2013.06.006

[4] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (Oct. 2017), 27 pages.

[5] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 54 (Oct. 2017), 24 pages.

[6] Michael Bayne, Richard Cook, and Michael D. Ernst. 2011. Always-available static and dynamic feedback. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE).* 521–530.

[7] Jon Bentley and Bob Floyd. 1987. Programming Pearls: A Sample of Brilliance. *Commun. ACM* 30, 9 (Sept. 1987), 754–757. https://doi.org/10.1145/30401.315746

[8] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings.* 257–281.

[9] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software.* ACM, New York, NY, 85–98.

[10] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. 2007. The development of the Emerald programming language. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007.* 1–51.

[11] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* 117–136.

[12] Gilad Bracha. 2004. Pluggable Type Systems. OOPSLA Workshop on Revival of Dynamic Languages. , 6 pages.

[13] Gilad Bracha and David Griswold. 1993. Stongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA.*

[14] Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. Seeking Grace: a new object-oriented language for novices. In *Proceedings 44th SIGCSE Technical Symposium on Computer Science Education.* ACM, 129–134.

[15] Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands.* 12:1–12:24. https://doi.org/10.4230/LIPIcs.ECOOP.2018.12

[16] Anderson B. N. da Silva, Daniel A. M. Cunha, Vitor R. G. Silva, Alex F. de A. Furtunato, and Samuel Xavier-de Souza. 2019. PaScal Viewer: A Tool for the Visualization of Parallel Scalability Trends. In *Programming and Performance Visualization Tools*, Abhinav Bhatele, David Boehme, Joshua A. Levine, Allen D. Malony, and Martin Schulz (Eds.). Springer International Publishing, Cham, 250–264.

[17] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'16).* ACM, 642–659.

[18] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

[19] Ben Greenman and Matthias Felleisen. 2018. A spectrum of type soundness and performance. *PACMPL* 2, ICFP (2018), 71:1–71:32. https://doi.org/10.1145/3236766

[20] Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-Tag Soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'18).* ACM, 30–39.

[21] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to evaluate the performance of gradual type systems. *Journal of Functional Programming* 29 (2019), 45. https://doi.org/10.1017/S0956796818000217

[22] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming (LNCS)*, Vol. 512. Springer, 21–38. https://doi.org/10.1007/BFb0057013

[23] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2014. State of the Art of Performance Visualization. In *EuroVis - STARs*, R. Borgo, R. Maciejewski, and I. Viola (Eds.). The Eurographics Association. https://doi.org/10.2312/eurovisstar.20141177

[24] Timothy Jones. 2017. *Classless Object Semantics.* Ph.D. Dissertation. Victoria University of Wellington.

[25] Timothy Jones, Michael Homer, James Noble, and Kim Bruce. 2016. Object Inheritance Without Classes. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Vol. 56. 13:1–13:26.

[26] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1993. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley.

[27] Stefan Marr. 2018. ReBench: Execute and Document Benchmarks Reproducibly. https://doi.org/10.5281/zenodo.1311762 Version 1.0.

[28] Stefan Marr. 2018. SOMns: A Newspeak for Concurrency Research. https://doi.org/10.5281/zenodo.3270908

[29] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16).* ACM, 120–131.

[30] Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 56 (Oct. 2017), 30 pages.

[31] E. Papenhausen, K. Mueller, M. H. Langston, B. Meister, and R. Lethin. 2016. An Interactive Visual Tool for Code Optimization and Parallelization Based on the Polyhedral Model. In *45th International Conference on Parallel Processing Workshops (ICPPW'16).* 309–318. https://doi.org/10.1109/ICPPW.2016.52

[32] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 55 (Oct. 2017), 27 pages.

[33] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

[34] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2017. Toward Virtual Machine Adaption Rather than Reimplementation. In *MoreVMs'17: 1st International Workshop on Workshop on Modern Language Runtimes, Ecosystems, and VMs at <Programming> 2017*. Presentation.

[35] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks are (Almost) Free. In *ECOOP*.

[36] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Seventh Workshop on Scheme and Functional Programming*, Vol. Technical Report TR-2006-06. University of Chicago, 81–92.

[37] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. 2–27.

[38] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 274–293.

[39] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*. 432–456.

[40] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Optimization Coaching: Optimizers Learn to Communicate with Programmers. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, 163–178. https://doi.org/10.1145/2384616.2384629

[41] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, 456–468.

[42] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. 793–810. https://doi.org/10.1145/2384616.2384674

[43] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 395–406.

[44] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 45–56.

[45] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. *CoRR* abs/1902.07808 (2019). arXiv:1902.07808 http://arxiv.org/abs/1902.07808

[46] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. ACM, 762–774.

[47] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *European Symposium on Programming Languages and Systems (ESOP)*. 1–16.

[48] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'14)*. ACM, 133–144.

[49] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, 662–676.

[50] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, 187–204.

[51] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium (DLS'12)*. 73–82. https://doi.org/10.1145/2384577.2384587