Naïve Transient Cast Insertion Isn't (That) Bad

ERIN GREENWOOD-THESSMAN, Victoria University of Wellington, New Zealand
ISAAC OSCAR GARIANO, Victoria University of Wellington, New Zealand
RICHARD ROBERTS, Victoria University of Wellington, New Zealand
STEFAN MARR, University of Kent, United Kingdom
MICHAEL HOMER, Victoria University of Wellington, New Zealand
JAMES NOBLE, Victoria University of Wellington, New Zealand

Transient gradual type systems often depend on type-based cast insertion to achieve good performance: casts are inserted whenever the static checker detects that a dynamically-typed value may flow into a statically-typed context. Transient gradually typed programs are then often executed using just-in-time compilation, and contemporary just-in-time compilers are very good at removing redundant computations.

In this paper we present work-in-progress to measure the ability of just-in-time compilers to remove redundant type checks. We investigate worst case performance, and so take a naïve approach, annotating every subexpression to inserting every plausible dynamic cast. Our results indicate that the Moth VM still manages to eliminate much of the overhead, by relying on the state-of-the-art SOMns substrate and Graal just-in-time compiler.

We hope these results will help language implementers evaluate the tradeoffs between dynamic optimisations (which can improve the performance of both statically and dynamically typed programs) and static optimisations (which improve only statically typed code).

Additional Key Words and Phrases: static, dynamic, gradual, Grace, Moth

1 INTRODUCTION

Gradual typing aims to support dynamic typing within static languages, increasing flexibility whilst maintaining some safety [Abadi et al. 1991], or alternatively to add static type annotations to dynamic languages, increasing their safety while maintaining flexibility [Bracha 2004; Siek and Taha 2006; Siek et al. 2015]. These two lineages [Chung et al. 2018; Greenberg 2019; Greenman and Felleisen 2018] of gradual typing lead to different implementation strategies: either extending the low-level back-end implementation of a static language to permit some dynamicity [Bierman et al. 2010; Kuhlenschmidt et al. 2019; Muehlboeck and Tate 2017], or extending the high-level front-end of a dynamic language to check type annotations before running programs on an (often unmodified) dynamic back-end implementation [Rastogi et al. 2015].

This paper takes a naïve approach to evaluating the effectiveness of a back-end just-in-time compiler to removing dynamic type checks. We will measure the performance of "transient" or "type-tag" checks (as in Reticulated Python), which offer first-order semantics: they check that an object's type constructor or names of supported methods match any static types that the object flows through, but not the return types or argument types of those methods [Bloom et al. 2009; Greenman and Migeed 2018; Richards et al. 2015; Siek and Taha 2007; Vitousek et al. 2014].

Authors' addresses: Erin Greenwood-Thessman, Engineering and Computer Science, Victoria University of Wellington, New Zealand, erin.greenwood-thessman@ecs.vuw.ac.nz; 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.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

We build on the work of Roberts et. al by building upon the open-source Moth VM based upon SOMns and the GraalVM [Gariano et al. 2019; Roberts et al. 2017, 2019] but we extend their work in one critical way. The Moth VM supports the Grace programming language semantics, which meet the "dynamic gradual guarantee" [Boyland 2014; Greenberg 2019] in that type annotations on declarations can be removed without changing a programs' behaviour. Grace does not meet the "refined gradual guarantee" however, because Grace checks types only when they flow through an explicit type annotation, and not e.g. when they flow into an intermediate expression. This means that the existing results for Moth [Roberts et al. 2019] may require fewer runtime type checks than a "sound" gradual system such as Reticulated Python [Vitousek et al. 2019]. To get a worst case estimate of the overhead of sound checking, we insert explicit type checks on *every* subexpression, so doing as many type checks as possible. In spite of this worst case, our results show that the performance is the same on average as when types are only checked when a value passes though an explicit type annotation.

The next section discusses dynamic type checks and gradual typing in Moth. Section 3 then describes our benchmarking protocol and Section 4 presents our results. Section 5 presents some additional related work, and finally section 6 concludes.

2 BACKGROUND

Our work is based on the Moth virtual machine [Roberts et al. 2017, 2019], an implementation of the Grace programming language [Black et al. 2012; Bruce et al. 2013]. Moth is based on the Graal and Truffle toolchain [Würthinger et al. 2017, 2013], and developed from a Newspeak implementation based on the Simple Object Machine [Marr 2018].

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 [Black et al. 2012; Bruce et al. 2013]. 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, and control structures are built out of lambda expressions supporting "non-local" returns from the lexically enclosing method. [Goldberg and Robson 1983]. In other ways, Grace is closer to JavaScript than Smalltalk: Grace objects are created from object literals, rather than by instantiating classes [Black et al. 2007; Jones et al. 2016], or Newspeak: Grace's objects and classes can be deeply nested within each other [Lehrmann Madsen et al. 1993].

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 [Boyland 2014]. 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 [Jones 2017]; here we explain how these types can be understood dynamically, from the Grace programmer's point of view. Grace's types are structural [Black et al. 2012], 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.

2.2 Moth's Transient Type Checking

Moth's implementation of transient type checks are only first-order. Thus, 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.

2.3 Moth's Optimisation

Like other VMs based on the Truffle and Graal toolchain, Moth is a self-optimising AST interpreter [Roberts et al. 2019; Würthinger et al. 2012]. 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" [Wößet al. 2014] 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" [Hölzle et al. 1991] 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 [Würthinger et al. 2017].

3 EXPERIMENTAL METHODOLOGY

Our experimental methodology is relatively simple: first we transform a benchmark suite by inserting all possible casts, and then we compare the performance of the transformed programs when run with and without type checking. We also compare performance with the untranslated benchmarks, again with and without type checks.

3.1 The Benchmarks

For this work, we rely on the benchmark suite compiled for previous work [Roberts et al. 2019]. It is a collection of 21 benchmarks in total, derived from the Are We Fast Yet benchmark suite [Marr et al. 2016] and other benchmarks from the gradual-typing literature.

Casts are manually added to versions of each of the benchmarks. We name these versions as *CastX*, where *X* is the name of benchmark. The casts are inserted to wrap method calls on objects with explicit receivers or when the call is an operation (like + or &).

Consider the following example where the get the last point value in some list:

```
1  def lastPos: Number = listOfPoints.size − 1
2
3  (list.size > 0).ifTrue {
4      lastPoint := listOfPoints.get(lastPos)
5  }
```

Assuming *listOfPoints* is known be of type *Array(Point)*, *.size* is a method that returns a number it should be wrapped by a cast, along with the subtraction of one (since we know the receiver is a number). The operator > for a boolean returns a boolean and should also be cast. Though Moth doesn't track type parameters, casts are inserted as if they were (so the results are still applicable for when Moth does support them). This means that the access of *listOfPoints* is also wrapped in a cast. This example would become:

```
1  def lastPos: Number = Number.cast(Number.cast(listOfPoints.size) − 1)
2
3  Boolean.cast(Number.cast(list.size) > 0).ifTrue {
4      lastPoint := Point.cast(listOfPoints.get(lastPos))
5  }
```

3.2 Benchmarking

To account for the complex warmup behaviour of modern systems [Barrett et al. 2017] as well as the non-determinism caused by e.g. garbage collection and cache effects, we ran each benchmark

for 300 iterations in the same invocation of Moth, and discard the first 10 iterations to ignore the worst of warmup JIT compilation.

Our experiment used a single machine with one Intel i7-8700 CPU running at 3.20GHz, with 6 cores for a total of 12 hyperthreads. The machine was running Arch Linux 5.1.12, and we used Java 1.8.0_212 Graal 19.0. Benchmarks were executed one by one to avoid interference between them. The analysis of the results and plots where generated using PGFPLOTS.

In previous work [Roberts et al. 2019], we also compared the performance of untyped code on Moth against state-of-the-art VMs: Java, Node.js using the V8 JavaScript VM, and the Higgs JavaScript VM. Java was the fastest of these, and on average V8 was about 1.8x slower than Java, Moth was 2.3x slower, and Higgs was 10.4x slower. We believe this makes Moth suitable for assessing the impact of type checking, because Moth's performance is close enough to state-of-the-art VMs, which should make it harder to hide type checking overheads in a slow baseline.

4 RESULTS

The results of running the benchmarks are shown in Figure 1. In all of the untyped executions, we could not detect a difference between the original and cast versions. This was expected as the casts should disappear entirely and produce the same machine code as the original version.

For the typed runs with casts, 15 of the 21 benchmarks had the same performance as the original version. Four benchmarks (CD, DeltaBlue, List, PyStone) performed worse with casts, and two sped up (GraphSearch, Richards).

Of the four benchmarks with a slowdown, the average slowdown was 6.18%. Those benchmarks (with percentage slowdown) are CD (10.34%), DeltaBlue (4.81%), List (3.73%), and PyStone (5.83%). For the two benchmarks with a speed up, GraphSearch was 6.40% faster and Richards was 38.03% faster.

Across the benchmark suite, adding casts did not change the overall performance significantly (an average speed up of 0.43%). If this result applies more generally, then it appears the existing Graal dynamic optimizer can be relied on to remove the overheads of the additional type checks, without any prior type-based static optimisation.

5 RELATED WORK

Other than our own studies [Gariano et al. 2019; Roberts et al. 2019], the most recent and most closely related work conducts very similar experiments, using Reticulated Python rather than Grace, and the PyPy VM rather than Moth [Vitousek et al. 2019]. This study finds similar results, that a JIT compiler can remove almost all the overhead of transient typing, even with a very naïve cast insertion strategy. This work then deploys an optimiser based on the static portion of Reticulated Python's gradual type system, so that type casts are only inserted if the type checker determines that they are required, further increasing performance.

A earlier study [Bauman et al. 2017] used Pycket [Bauman et al. 2015] (a tracing JIT for Racket) rather than the standard Racket VM, but maintained Racket's full gradually-typed semantics while using object shapes to encode information about gradual types. This approach demonstrated most benchmarks ran well, although with a slowdown of 2x on average over all configurations — significantly worse than the Reticulated Python results, or our results. Note that since typed modules do not need to do any checks at run time, Typed Racket only needs to perform checks at boundaries between typed and untyped modules — effectively a . static, type based optimisations, again like Reticulated Python but unlike our naïve approach.

The Nom language [Muehlboeck and Tate 2017] was specifically designed to make gradual types easier to optimize, demonstrating speedups as more type information is added to programs. Their approach enables such type-driven optimizations, but relies on a static analysis which can utilize

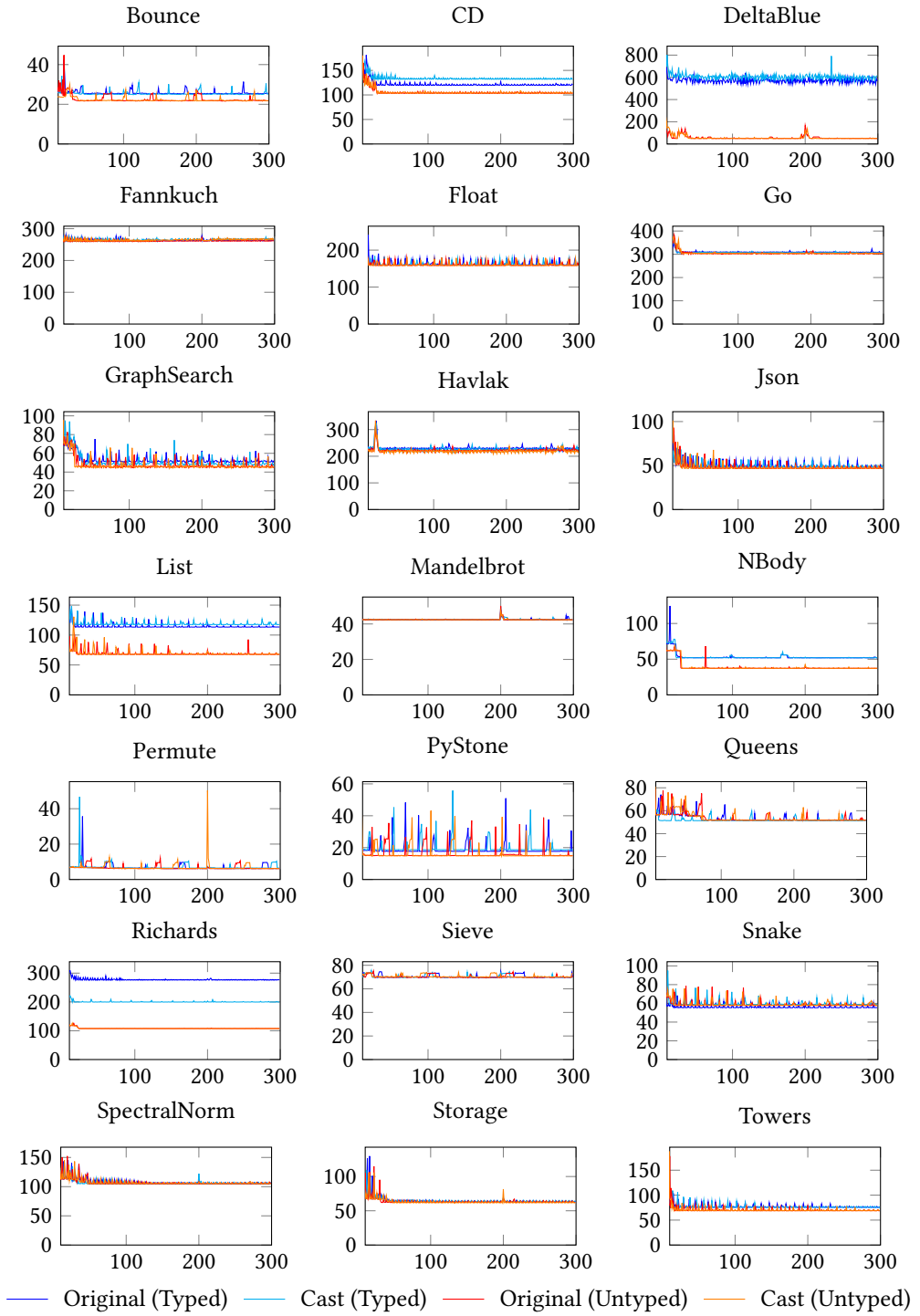

Fig. 1. Benchmark performance. Original is the benchmark version without the inserted casts and Cast is the version with the casts. Vertical axis is execution time (ms) and horizontal axis is number of iterations

the type information, and the underlying types are nominal, rather than structural. Similarly, the Grift language [Kuhlenschmidt et al. 2019] is designed to take advantage of a traditional, ahead of time, static compiler, and demonstrates good performance for code where more than half of the program is annotated with types, and reasonable performance for code without type annotations.

The SafeTypeScript language has also been implemented by extending the Higgs VM [Richards et al. 2017], although implementing "monotonic" gradual typing with blame, rather than the simpler transient checks used in Moth and Reticulated Python. This study again demonstrates that JIT-ing VMs can eliminate most of the overhead of dynamic type checks, although the overall performance is significantly slower than the other approaches.

6 CONCLUSION

In this paper we have measured the performance impact of naïvely inserting every possible (sensible) dynamic cast into a gradually typed program, and then running that program on Moth, a VM for the Grace language based on Truffle and Graal.

We found that on average the performance of the cast-inserted benchmarks was the same as only type checking at explicit type annotations. Some benchmarks were somewhat slower, and one benchmark (Richards) was almost twice as fast. Our next research goal is to try to understand the cause of these changes.

Finally, we hope these results will help language implementers evaluate the tradeoffs between dynamic optimisations (which can improve the performance of both statically and dynamically typed programs) and static optimisations (which improve only statically typed code).

ACKNOWLEDGMENTS

This work is supported in part by the Royal Society of New Zealand (Te Apārangi) Marsden Fund (Te Pūtea Rangahau a Marsden) under grant VUW1815.

REFERENCES

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.

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.

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 22–34.

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.

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *ECOOP*.

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.

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.

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.

John Tang Boyland. 2014. The Problem of Structural Type Tests in a Gradual-Typed Language. In *FOOL*.

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

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.

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

Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Which of my Transient Type Checks are not (Almost) Free?. In *VMIL*.

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

Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *SNAPL (LIPIcs)*, Vol. 136.

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

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.

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

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

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.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward efficient gradual typing for structural types via coercions. In *PLDI*.

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

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

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.

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.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 167–180.

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.

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

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.

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

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.

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.

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.

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.

Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. In *DLS*.

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.

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.

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.

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