Naïve Transient Cast Insertion Isn't (That) Bad

Erin Greenwood-Thessman
School of Engineering and Computer Science
Victoria University of Wellington
New Zealand
erin.greenwood-thessman@ecs.vuw.ac.nz

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

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

ABSTRACT

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 insert 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).

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; **Data types and structures**.

KEYWORDS

static, dynamic, gradual, Grace, Moth

ACM Reference Format:
Erin Greenwood-Thessman, Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2021. Naïve Transient Cast Insertion Isn't (That) Bad. In *Proceedings of the 16th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS '21), July 13, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3464972.3472395

ICOOOLPS '21, July 13, 2021, Virtual, Denmark

1 INTRODUCTION

Gradual typing aims to support dynamic typing within static languages, increasing flexibility whilst maintaining some safety [1], or alternatively to add static type annotations to dynamic languages, increasing their safety while maintaining flexibility [10, 37, 39]. These two lineages [12, 15, 16] of gradual typing lead to different implementation strategies: either extending the low-level back-end implementation of a static language to permit some dynamicity [5, 26, 30], 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 [32].

This paper takes a naïve approach to evaluating the effectiveness of a back-end just-in-time compiler in removing dynamic type checks. We will measure the performance of "transient" or "type-tag" checks (as in Reticulated Python), which inspect objects when they pass through local type declarations (for example, at assignments and method calls) to check that the objects' type constructors or names of supported methods match those expected at that point. Transient checks impose no ongoing obligation to continue to satisfy the type, nor require full information about an object or type. Instead, whenever a method is called on an object within the lexical scope of a given type annotation, the parameter and return types must be checked in accordance with that annotation. Because a dynamically-typed method may return any value those checks are necessarily first-order. Transient checks thus pertain to the variable within its scope, and not to the object or the reference, ensuring local soundness and that type errors can be blamed on outside code [8, 17, 34, 38, 41].

We build on the work of Roberts et. al by building upon the open-source Moth VM (based upon SOMns and the GraalVM) [13, 35, 36], but we extend their work in one critical way. While Moth performs first-order type checks at assignments and when emitting a return value from a method, confirming that the value satisfies the type

annotation on that location, it does not inspect anonymous intermediate values that are never stored in an annotated variable, field, or parameter, such as those returned from a method and discarded, or in "chained" method calls. While these return values must satisfy the type declared in the method definition, if any, they are not checked for compatibility with the type declared locally for the receiver. These semantics meet the "dynamic gradual guarantee" [9, 15], in that type annotations on declarations can be removed without changing a program's behaviour, but not the "refined gradual guarantee" [40], because a violation of the locally-expected return type of a method may be unnoticed. This means that the existing results for Moth [36] may require fewer runtime type checks than a "sound" gradual system such as Reticulated Python [42]. To get a worst-case estimate of the overhead of sound checking, we insert explicit type checks on *every* subexpression, in so doing as many type checks as possible. In spite of this worst case scenario, our results show that the performance is the same on average as when types are only checked when a value passes through an explicit type annotation, which [36] found to be negligible.

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 6 presents some additional related work, and finally section 7 concludes.

2 BACKGROUND

Our work is based on the Moth virtual machine [35, 36], an implementation of the Grace programming language [6, 11]. Moth is based on the Graal and Truffle toolchain [44, 45], and developed from a Newspeak implementation based on the Simple Object Machine [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 [6, 11]. 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. [14]. In other ways, Grace is closer to JavaScript than Smalltalk: Grace objects are created from object literals, rather than by instantiating classes [7, 25], or Newspeak: Grace's objects and classes can be deeply nested within each other [27].

Grace's Typing. In Grace, all declarations can be annotated with types. As Grace is designed to support a variety of teaching methods, implementations 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 [9]. 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 [23]; here we explain how these types can be understood dynamically, from the Grace programmer's point of view. Grace's core types are structural [6]; 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, when testing whether an object value satisfies a type annotation, Moth only checks dynamically that the object has methods of the same name and arity as are required by the type: any argument and return types of such methods are not compared.

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, which 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 variable's declaration.

Under the Grace object model, object fields are included in the above without requiring further specification, because they are accessed only through getter and setter accessor methods on their containing object, as are parameters to lambda blocks and class constructors, which are ordinary method parameters.

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, not shown) within the definition of printNumber(n : Number) , Moth will again check that the value is a Number.

Note that we never check either whether the result of requesting "t.three" is actually a String (as one may expect from line 5), nor whether the object o's "three" methods is *expected* to return a String, 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.

This depth of checking does not implement the full semantics of Grace's structural types, but on a practical level catches a wide range of real-world errors, and so has been regarded as a "good enough" intermediate step in existing Grace implementations. Our work here is the first step in extending Moth to the full higher-order checks of the language semantics, by inspecting all concrete values for first-order conformance to their locally-expected types.

2.3 Moth's Optimisation

Like other virtual machines based on the Truffle and Graal toolchain, Moth is a self-optimising Abstract Syntax Tree (AST) interpreter [36, 46]. 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 an object-oriented dynamic dispatch and exposing optimization opportunities to the just-in-time compiler.

Moth relies on a number of standard techniques for optimising object-oriented programs. "Shapes" [43] 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 [44].

For the case of run-time type checking in particular, Moth will be able to eliminate redundant checks and utilise information already held by the virtual machine, but not necessarily apparent in the code, to optimise their execution. An explicit type check may also make certain information available earlier or for longer, and so enable further general optimisations in the generated machine code. For example, although the language may not guarantee that a field value has not changed, inlining may have ensured that the type does not change and so all repeated checks can be removed.

Roberts et al. [36] found that Moth's baseline performance was within 31% of the Node.js (V8 JavaScript) runtime without type checking, and that type-checking of non-intermediate values had an average 5% overhead. That work details the core optimisations of Moth itself in more detail.

3 EXPERIMENTAL METHODOLOGY

Our experimental methodology is relatively simple: first we transform a benchmark suite by inserting all possible casts manually, 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.

A fully-annotated program can be seen as analogous to transforming the existing program into static-single-assignment form, reifying all intermediate values into explicit local variables, and propagating all necessary type annotations manually. The program from Section 2.2 could be rendered as follows:

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

In this case, the assignment to "tmp" would detect that a number did not have all the methods of String and report a type error, finding a flaw not detected with the original program. Creating these additional local variables would itself be a confound, however, so our benchmark suite inserts explicit dynamic checks instead (for example, the final line above would instead become printNumber(String.cast(t.three))).

These fully-annotated programs can have many more run-time type checks to perform, sometimes by orders of magnitude. They are gradually sound up to parameter and return types of unused methods: it is not possible to receive a no-such-method exception on a call chain starting from a typed value, unless the declared type includes an explicit Unknown.

3.1 The Benchmarks

For this work, we rely on the benchmark suite compiled for previous work [36]. 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.

For each benchmark, we manually added explicit cast operations at every possible location. We named these versions as *CastX*, where *X* is the name of benchmark. The casts are inserted to wrap method calls both on objects with explicit receivers (like o.three) and when the call is an operation (like + or &).

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

```
def lastPos: Number = listOfPoints.size − 1

(list.size > 0).ifTrue {
    lastPoint := listOfPoints.get(lastPos)
}
```

Assuming *listOfPoints* is known to 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 number returns a boolean and should also be cast. Though Moth does not 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:

```
def lastPos: Number =
  Number.cast(Number.cast(listOfPoints.size) − 1)

Boolean.cast(Number.cast(list.size) > 0).ifTrue {
    lastPoint := Point.cast(listOfPoints.get(lastPos))
}
```

These cast operations perform the type check and raise an exception if it is unsatisfied. As dynamic operations, they do not expand the stack size of the method the way that additional local variables would, and so do not have unnecessary side effects with the Moth or Truffle optimisers. They do not perform any transformation of the values (e.g. a cast from a number to a boolean is an error), but only inspect the concrete objects independent of their existing type annotations (so an object that happens to satisfy both types, even if they are unrelated, will not trigger an error).

3.2 Benchmarking

To account for the complex warmup behaviour of modern systems [2] 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 were generated using PGFPLOTS.

In previous work [36] 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 2. 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. Figure 1 shows the average performance of all benchmarks and the measured overhead of the version with inserted casts.

Benchmark	Original	Casts	Overhead
Towers	77.00	76.78	-0.29%
Storage	64.43	64.38	-0.07%
SpectralNorm	107.08	106.63	-0.43%
Snake	56.16	59.88	+6.62%
Sieve	70.55	70.34	-0.30%
Richards	277.54	200.30	-27.83%
Queens	53.44	51.89	-2.90%
PyStone	18.81	19.75	+4.99%
Permute	6.79	7.11	+4.66%
NBody	53.33	53.15	-0.32%
Mandelbrot	42.45	42.51	+0.14%
List	114.59	118.82	+3.69%
Json	49.84	49.69	-0.29%
Havlak	230.10	227.29	-1.22%
GraphSearch	53.23	51.18	-3.85%
Go	309.90	307.70	-0.71%
Float	162.03	161.54	-0.30%
Fannkuch	263.03	264.96	+0.73%
DeltaBlue	572.22	604.87	+5.71%
CD	121.87	133.96	+9.92%
Bounce	25.75	25.43	-1.24%
Overall	2730.14	2698.16	-1.17%
Average	-	-	-0.15%

Figure 1: Mean execution time (ms) of each benchmark, ignoring 10 warmup runs, with overhead compared to type-checked execution of benchmark without cast insertions. Overall represents total average runtime of benchmark suite, while Average is the mean of benchmark overheads.

For the typed runs with casts, 13 of the 21 benchmarks had the same performance as the original version. Six benchmarks (CD, DeltaBlue, List, PyStone, Permute, Snake) performed worse with casts, and three sped up (GraphSearch, Richards, Queens).

Of the six benchmarks with a slowdown, the average slowdown was 5.93%. Those benchmarks (with percentage slowdown) are CD (9.92%), DeltaBlue (5.71%), List (3.69%), PyStone (4.99%), Snake (6.62%), and Permute (4.66%)). For the three benchmarks with a speed up, GraphSearch was 3.85% faster, Richards was 27.83% faster, and Queens was 2.90% faster. In total, 13 benchmarks were measured as at least slightly faster, and eight as at least slightly slower, but most of the differences are extremely small and some are likely noise.

Across the benchmark suite, adding casts did not change the overall performance significantly (total speed up of 1.17%, averaging 0.15% per benchmark). 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.

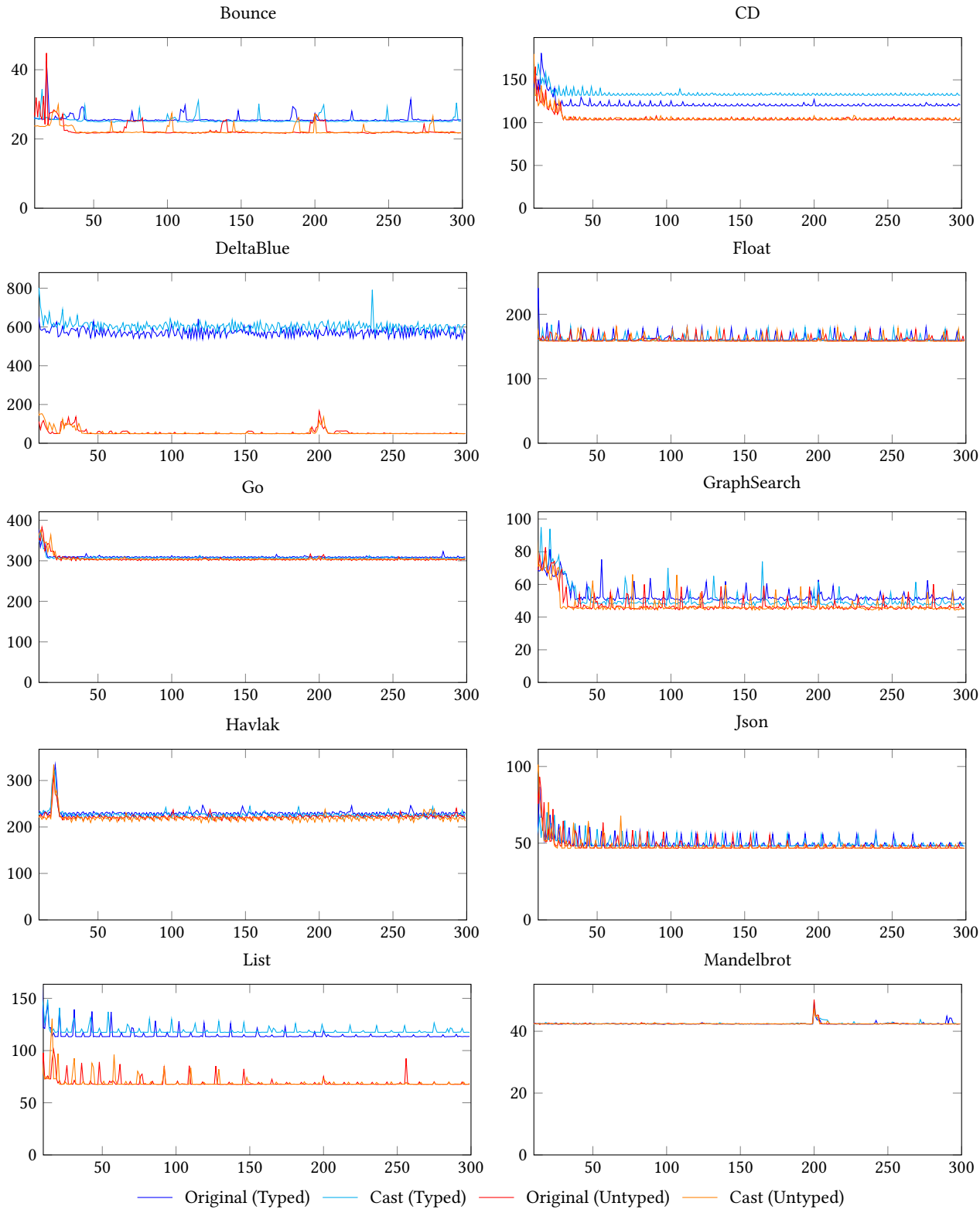

Figure 2: Benchmark performance. Original is the benchmark version without inserted casts, and Cast is with casts. Untyped is with type checking disabled in the VM, without modifying source code. Y axis is execution time (ms) and X is iterations.

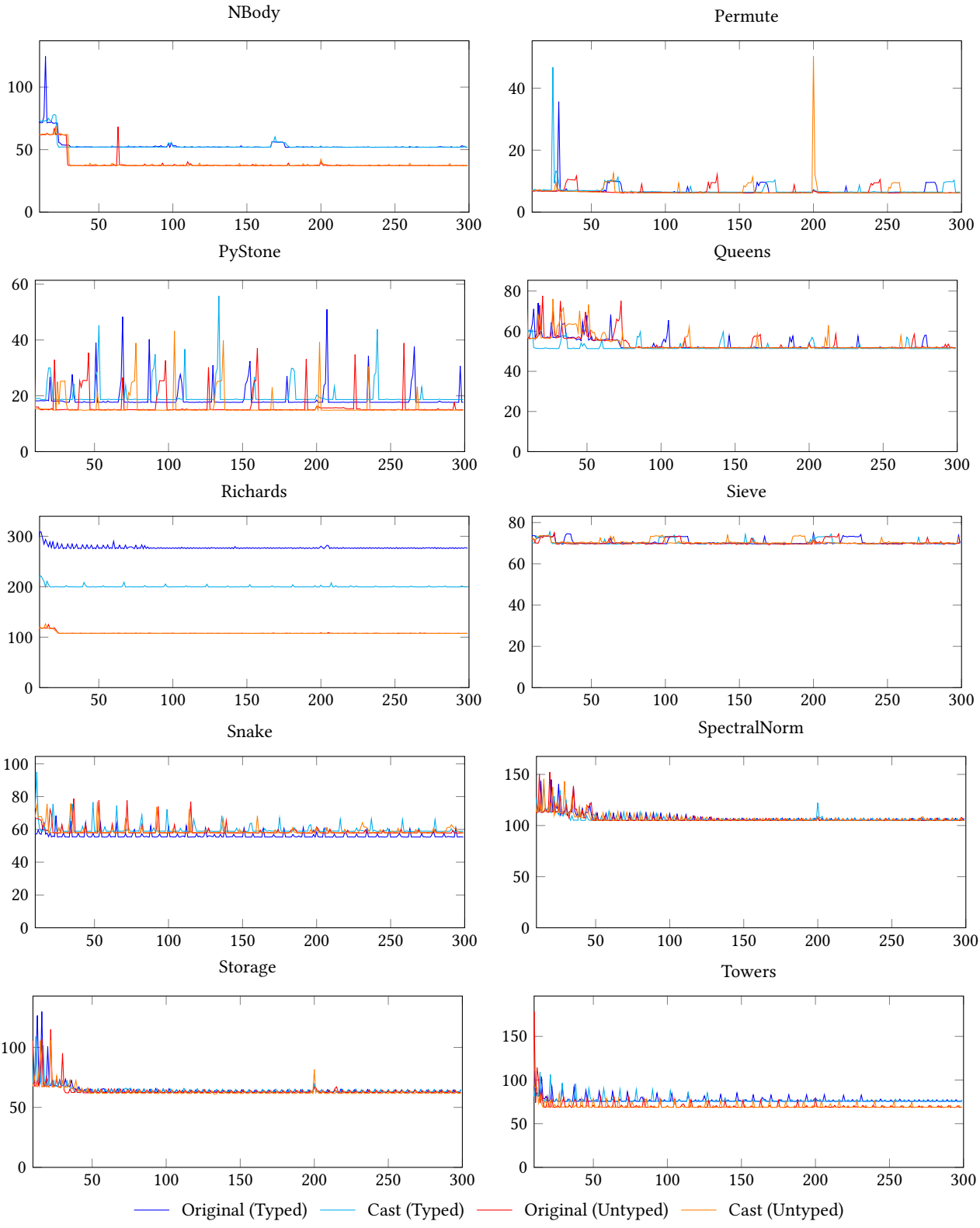

Figure 2 continued.

The Richards benchmark is one Roberts et al. [36] reported as having significantly *worse* performance with type annotations on. Our results continue to show overhead compared to untyped, but the improvement from additional type checks is substantial. Roberts et al. identified a particular sort of linked-list traversal within the benchmark as a pathological case for this type of optimiser, because it results in a check that *does not* correspond to information the virtual machine already holds. The additional checks for intermediate phases appear to bridge much of this gap: more information is available locally to be reused, and the resulting machine code includes fewer redundant checks. Gariano et al. [13] found that certain type annotations within the baseline Richards benchmark were significant: one specific type annotation caused significant further performance degradation when it was present, unless a certain other was also present. As our benchmark includes both (and more), we do not believe this specific complication should be affecting the results. However, given the sizable performance improvement it is likely that there are further such pairs where the "good" type test does not correspond to one explicit in the surface syntax of the base program, and our cast insertion has counterbalanced some other expensive tests from the original benchmark. Further investigation of this interesting case is ongoing.

The Queens benchmark appears to stabilise much more quickly in the casts condition than any other, with the original and untyped versions taking nearly ten times as long to reach steady-state performance, which is then similar across all conditions. In the graphs of Roberts et al. [36] this benchmark also shows a lengthy stabilisation period. This is an interesting result not seen in other benchmarks and suggests that intermediate types help to identify run-time optimisations quickly in this case, despite eventual performance being similar.

The performance changes for the other benchmarks are much smaller than Richards, but not insignificant. CD, Snake, List, and DeltaBlue were reported by Roberts et al. [36] as also slower with types enabled, while GraphSearch similarly showed improved performance. PyStone and Permute originally had somewhat *improved* performance, contrary to our results, though these are the two briefest benchmarks. These cases also merit further investigation to determine what factors influence these tendencies. The smaller changes may simply be noise particular to the specifics of the benchmark, such as shifting the boundaries of compilation units in the optimiser, or may represent real difference.

5 DISCUSSION

5.1 Future Work

Two main pieces of future work arise from this study. First is to extend Moth to propagate these local type annotations mechanically; this would not bring Moth to the full higher-order checking of parameter and return types of unused methods, but would be a step along the path to full gradual typing. We now know from this work that the performance implications of naïve cast insertion are acceptable, which permits a straightforward approach. At this point further benchmarking would be possible, and exploration of additional steps for VM optimisation to find any that still have benefit on top of this baseline naïve approach.

The second piece of work is to investigate further in which kinds of scenario these intermediate checks create performance changes, positive or negative, with an eye to improved virtual machine optimisation to take advantage of, or mitigate, these situations. These investigations may follow the model of Gariano et al. [13], but incorporating intermediate typechecks as individually-toggleable facets as well. In particular, intermediate values that are never saved at all may have distinct characteristics for performance measurements.

Because Grace is structurally-typed, *all* (non-trivial) types in Moth are higher-order, so the required degree of checking is much increased compared to a nominally-typed language where type names and implementations are closely connected. Nominal checks *without* such coupling, as in the brands design previously proposed for Grace [24], present a different set of challenges, as they also do not correspond to information the virtual machine inherently has on hand. Consequently, it is not immediately clear whether these type checks would also be "free". Further studies should incorporate brand types in some benchmarks to determine the implications.

The Grace language also admits further type extensions through user code, both dynamically through pattern-matching [19, 21] and statically through "dialects" [20]. Arbitrary dynamic types in particular may be amenable to exactly the same optimisations we see in this work, though they do not correspond as cleanly to information the VM already knows. The implications of our result to such type-system extensions is uncertain, and additional studies could determine the bounds of the "almost free" checking seen so far. Other Grace extensions to inheritance [25, 31] or method resolution [18], which each directly affect the (purported) shapes of objects, may also have either no, or significant, performance implications, and until further study is completed it is not obvious which.

While it is clear from this study, and those of Roberts et al. [36] and Gariano et al. [13], that a naïve approach to checking dynamic types in a modern optimising virtual machine is adequate as a baseline, and so the previous assumption[1] that adding dynamically-checked types to a program would necessarily cause a performance loss is unnecessary, there remain many further optimisation steps that could be taken. It is possible that existing optimisations taken by other systems may produce further advantage layered on top, but also possible that those systems could better adopt this simple approach using commodity components, rather than maintaining bespoke optimisers, without losing soundness or performance.

5.2 Threats to Validity

This work is subject to many of the common threats to validity of experimental works. Although our benchmarks appear to have stabilised, it is possible that further executions would reach another breakpoint (as are seen earlier in some benchmarks) that separated the conditions further, or brought steady-state performance to equality with a longer warmup period. Contrariwise, our results also show variation in how long stabilisation takes in different

[1] For example, Chung et al. [12] found that "The transient approach checks types at uses, so the act of adding types to a program introduces more casts and may slow the program down (even in fully typed code)." and say "transient semantics...is a worst case scenario... , there is a cast at almost every call", while Greenman and Migeed [17] found a "clear trend that adding type annotations adds performance overhead. The increase is typically linear."

conditions, and it is arguable that excluding warmup is obscuring material information; we may be examining (partly) the wrong thing. Our augmented benchmarks may have introduced inaccurate type checks, threatening construct validity. Moth itself may have bugs that affect both construct and internal validity, and we do not have sufficient information to say that our findings generalise to virtual machines following a different model to Moth, threatening external validity. Other languages have additional constructs not included in Grace (and vice-versa, though the benchmarks we are using strove to counter that), and such constructs may introduce additional threats to generalisability of our results by making type tests more expensive or frequent. Finally, we may have errors in our analysis of our results, either human or machine.

6 RELATED WORK

Other than the earlier Moth studies [13, 36], 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 [42]. 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 [4] used Pycket [3] (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 optimisation, again like Reticulated Python but unlike our naïve approach.

The Nom language [30] 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 the type information, and the underlying types are nominal, rather than structural. Similarly, the Grift language [26] 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 [33], 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.

7 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

[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. https://doi.org/10.1145/103135.103138

[2] 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. https://doi.org/10.1145/3133876

[3] 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. https://doi.org/10.1145/2784731.2784740

[4] 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. https://doi.org/10.1145/3133878

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

[6] 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. https://doi.org/10.1145/2384592.2384601

[7] 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. https://doi.org/10.1145/1238844.1238855

[8] 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. https://doi.org/10.1145/1639949.1640098

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

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

[11] 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. https://doi.org/10.1145/2445196.2445240

[12] 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

[13] 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*.

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

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

[16] 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

[17] 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* (Los Angeles, CA, USA) *(PEPM'18)*. ACM, 30–39. https://doi.org/10.1145/3162066

[18] Michael Homer, Timothy Jones, and James Noble. 2015. From APIs to Languages: Generalising Method Names. In *Dynamic Language Symposium*. https://doi.org/10.1145/2816707.2816708

[19] Michael Homer, Timothy Jones, and James Noble. 2019. First-Class Dynamic Types. In *Dynamic Language Symposium*. https://doi.org/10.1145/3359619.3359740

[20] Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. 2014. Graceful Dialects. In *European Conference on Object-Oriented Programming*. https://doi.org/10.1007/978-3-662-44202-9_6

[21] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns as Objects in Grace. In *Dynamic Language Symposium*.

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

[24] Timothy Jones, Michael Homer, and James Noble. 2015. Brand Objects for Nominal Typing. In *European Conference on Object-Oriented Programming*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.198

[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. https://doi.org/10.4230/LIPIcs.ECOOP.2016.13

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

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

[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. https://doi.org/10.1145/3093334.2989232

[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. https://doi.org/10.1145/3133880

[31] James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Timothy Jones. 2017. Grace's Inheritance. *The Journal of Object Technology* Volume 16, no. 2 (2017).

[32] 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. https://doi.org/10.1145/2676726.2676971

[33] 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. https://doi.org/10.1145/3133879

[34] 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

[35] 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* (Brussels, Belgium). Presentation.

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

[37] 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.

[38] 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. https://doi.org/10.1007/978-3-540-73589-2_2

[39] 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), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[40] 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, May 3-6, 2015, Asilomar, California, USA*. 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[41] 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. https://doi.org/10.1145/2661088.2661101

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

[43] 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* (Cracow, Poland) *(PPPJ'14)*. ACM, 133–144. https://doi.org/10.1145/2647508.2647517

[44] 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* (Barcelona, Spain) *(PLDI'17)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[45] 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* (Indianapolis, Indiana, USA) *(Onward! 2013)*. ACM, 187–204. https://doi.org/10.1145/2509578.2509581

[46] 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* (Tucson, Arizona, USA) *(DLS'12)*. 73–82. https://doi.org/10.1145/2384577.2384587