Transient Typechecks are (Almost) Free

Richard Roberts
School of Design, Victoria University of Wellington
rykardo.r@gmail.com

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

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

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

Abstract

Transient gradual typing imposes run-time type tests that typically cause a linear slowdown in programs' performance. This performance impact discourages the use of type annotations because adding types to a program makes the program slower. A virtual machine can employ standard just-in-time optimizations to reduce the overhead of transient checks to near zero. These optimizations can give gradually-typed languages performance comparable to state-of-the-art dynamic languages, so programmers can add types to their code without affecting their programs' performance.

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Object oriented languages; Software and its engineering → Interpreters

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

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.15

Funding This work is supported by the Royal Society of New Zealand Marsden Fund

1 Introduction

> *"It is a truth universally acknowledged, that a dynamic language in possession of a good user base, must be in want of a type system."*
>
> with apologies to Jane Austen.

Dynamic languages are increasingly prominent in the software industry. Building on the pioneering work of Self [20], much work in academia and industry has gone into making them more efficient [13, 14, 66, 24, 23, 25]. Just-in-time compilers have, for example, turned JavaScript from a naïvely interpreted language barely suitable for browser scripting, into a highly efficient ecosystem, eagerly adopted by professional programmers for a very wide range of tasks [44].

A key advantage of these dynamic languages is the flexibility offered by the lack of a static type system. From the perspective of many computer scientists, software engineers, and computational theologists, this flexibility has the disadvantage that programs without types are more difficult to read, to understand, and to analyze than programs with types. Gradual Typing aims to remedy this disadvantage, adding types to dynamic languages while maintaining their flexibility [16, 48, 50].

There is a spectrum of different approaches to gradual typing [22, 28]. At one end — "pluggable types" as in Strongtalk [17] or "erasure semantics" as in TypeScript [8] — all types are erased

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 15; pp. 15:1–15:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

before the execution, limiting the benefit of types to the statically typed parts of programs, and preventing programs from depending on type checks at run time. In the middle, "transient" or "type-tag" checks as in Reticulated Python offer first-order semantics, checking whether an object's type constructor or supported methods match explicit type declarations [49, 11, 46, 60, 29]. Reticulated Python also supports an alternative "monotonic" semantics which mutates an object to narrow its concrete type when it is passed into a more specific type context. At the other end of the spectrum, behavioral typechecks as in Typed Racket [59, 57], Gradualtalk [3], and Reticulated Python's proxies, support higher-order semantics, retaining types until run time, performing the checks eagerly, and giving detailed information about type violations as soon as possible via blame tracking [63, 2]. Finally, Ductile typing dynamically interprets a static type system at runtime [7]. Unfortunately, any gradual system with run-time semantics (i.e. everything more complex than erasure) currently imposes a significant run-time performance overhead to provide those semantics [56, 62, 42, 6, 45, 55, 29, 30].

The performance cost of run-time checks is problematic in itself, but also creates perverse incentives. Rather than the ideal of gradually adding types in the process of hardening a developing program, the programmer is incentivized to leave the program untyped or even to *remove* existing types in search of speed. While the Gradual Guarantee [50] requires that removing a type annotation does not affect the result of the program, the performance profile can be drastically shifted by the overhead of ill-placed checks. For programs with crucial performance constraints, for new programmers, and for gradual language designers, juggling this overhead can lead to increased complexity, suboptimal software-engineering choices, and code that is harder to maintain, debug, and analyze.

In this paper, we focus on the centre of the gradual typing spectrum: the transient, first-order, type-tag checks as used in Reticulated Python and similar systems. Several studies have found that these type checks have a negative impact on programs' performance. Chung, Li, Nardelli and Vitek, for example, 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"*" [22]. Greenman and Felleisen find that the slowdown is predictable, as transient checking "*imposes a run-time checking overhead that is directly proportional to the number of [type annotations] in the program"*" [28], and Greenman and Migeed found a "*clear trend that adding type annotations adds performance overhead. The increase is typically linear.*" [29].

In contrast, we demonstrate that transient type checks can be "almost free" via a just-in-time compiler to an optimizing virtual machine. We insert gradual checks naïvely, for each gradual type annotation. Whenever an annotated method is called or returns, or an annotated variable is accessed, we check types dynamically, and terminate the program with a type error if the check fails. Despite this simplistic approach, a just-in-time compiler can eliminate redundant checks—removing almost all of the checking overhead, resulting in a performance profile aligned with untyped code.

We evaluate our approach by adding transient type checks to Moth, an implementation of the Grace programming language built on top of Truffle and the Graal just-in-time compiler [67, 66]. Inspired by Richards *et al.* [45] and Bauman *et al.* [6], our approach conflates types with information about the dynamic object structure (maps [20] or object shapes [65]), which allows the just-in-time compiler to reduce redundancy between checking structure and checking types; consequently, most of the overhead that results from type checking is eliminated.

The contributions of this paper are:

- demonstrating that VM optimizations enable transient gradual type checks with low performance cost
- an implementation approach that requires only small changes to existing abstract-syntax-tree interpreters
- an evaluation based on classic benchmarks and benchmarks from the literature on gradual typing

2 Gradual Types in Grace

This section introduces Grace, and motivates supporting transient gradual typing in the language.

2.1 The Grace Programming Language

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, 19]. While Grace's syntax draws from the so-called "curly bracket" tradition of C, Java, and JavaScript, the structure of the language is in many ways closer to Smalltalk: all computation is via dynamically dispatched "method requests" where the object receiving the request decides which code to run, and returns within lambdas that are "non-local", returning to the method activation in which the block is instantiated [27]. In other ways, Grace is closer to JavaScript than Smalltalk: Grace objects can be created from object literals, rather than by instantiating classes [10, 35] and objects and classes can be deeply nested within each other [37].

Critically, Grace's declarations and methods' arguments and results can be annotated with types, and those types can be checked either statically or dynamically. This means the type system is intrinsically gradual: type annotations should not affect the semantics of a correct program [50], and the type system includes a distinguished "`Unknown`" type which matches any other type and is the implicit type for untyped program parts.

The static core of Grace's type system is well described elsewhere [34]; 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 implements a type whenever it implements all the methods required by that type, rather than requiring classes or objects to declare types explicitly. Methods match when they have the same name and arity: argument and return types are ignored. A type thus expresses the requests an object can respond to, for example whether a particular accessor is available, rather than a nominal location in an explicit inheritance hierarchy.

Grace then checks the types of values at run time:

- the values of arguments are checked after a method is requested, but before the body of the method is executed;
- the value returned by a method is checked after its body is executed; and
- the values of variables are checked whenever written or read by user code.[1]

In the spectrum of gradual typing, these semantics are closest to the transient typechecks of Reticulated Python [60, 29]. Reticulated Python inserts transient checks only when a value

[1] Checking on read in addition to writes may seem unnecessary. For the rational, see Section 6.2.

flows from untyped to typed code, while Grace inserts transient checks only at explicit type annotations (but in principle checks every annotation every time).

2.2 Why Gradual Typing?

Our primary motivation for this work is to provide a flexible system to check consistency between an execution of a program and its type annotations. A key part of the design philosophy of Grace is that the language should not force students to annotate programs with types until they are ready, so that teachers can choose whether to introduce types, early, late, or even not at all.

A secondary goal is to have a design that can be implemented with only a small set of changes to facilitate integration in existing systems.

Both of these goals are shared with much of the other work on gradual type systems, but our context leads to some different choices. First, while checking Grace's type annotations statically may be optional, checking them dynamically should not be: any value that flows into a variable, argument, or result annotated with a type must conform to that type annotation. Second, adding type annotations should not degrade a program's performance, or rather, programmers should not be encouraged to improve performance by removing type annotations. And third, we allow the programmer to execute a program even when not statically type-correct. Allowing such execution is useful to students, where they can see concrete examples of dynamic type errors. This is possible because Grace's static type checking is optional, which means that an implementation cannot depend on the correctness or mutual compatibility of a program's type annotations.

Unfortunately, existing gradual type implementations do not meet these goals, particularly regarding performance; hence the ongoing debate about whether gradual typing is alive, dead, or some state in between [56, 62, 42, 6, 45, 29, 30].

2.3 Using Grace's Gradual Types

We now illustrate how the gradual type checks work in practice in the context of a simple program to record information about vehicles. Suppose the programmer starts developing this vehicle application by defining an object intended to represent a car (Listing 1, Line 1) and writes a method that, given the car object, prints out its registration number (Line 5).

```
1 def car = object {
2     var registration is public := "JO3553"
3 }
4
5 method printRegistration(v) {
6     print "Registration: {v.registration}"
7 }
```

Listing 1 The start of a simple Grace program for tracking vehicle information.

Next, the programmer adds a check to ensure any object passed to the `printRegistration` method will respond to the `registration` request; they define the structural type `Vehicle` [58] naming just that method (Listing 2, Line 1), and annotate the `printRegistration` method's argument with that type (Listing 2, Line 5). The annotation ensures that a type error will be thrown if an object, passed to the `printRegistration` method,

```
type Vehicle = interface {
    registration
}

method printRegistration(v: Vehicle) {
    print "Registration: {v.registration}"
}
```

Listing 2 Adding a type annotation to a method parameter.

cannot respond to the `registration` message. Without the type check, the `print` method would cause a run-time error when interpolating the string. However, since type errors cause termination, the run-time error in the middle of the `print` implementation will now be avoided.

In Listing 3, the programmer continues development and creates two car objects (Lines 9 and 17), that conform to an expanded `Vehicle` type (Line 1).

```
1  type Vehicle = interface {
2      registration
3      registerTo(_)
4  }
5
6  type Person = interface { name }
7  type Department = interface { code }
8
9  var personalCar : Vehicle :=
10   object {
11     var registration is public := "DLS018"
12     method registerTo(p: Person) {
13       print "{p.name} registers {self}"
14     }
15   }
16
17 var governmentCar : Vehicle :=
18   object {
19     var registration is public := "FKD218"
20     method registerTo(d: Department) {
21       print "some department {self}"
22     }
23   }
24
25 governmentCar.registerTo(
26   object {
27     var name is public := "Richard"
28   }
29 )
```

Listing 3 A program in development with inconsistently typed `registerTo` methods.

Note that each version of the `registerTo` method declares a different type for its parameter (Lines 12 and 20). When the programmer executes this program, both `personalCar` and `governmentCar` can be assigned to a variable declared as `Vehicle` because checking that assignment considers only that the vehicle has a `registerTo` method, but not the required argument type of that method. At Line 25 the developer attempts to register a government car to a person: only when the method (Line 20) is *invoked* will the gradual type test on the argument fail (the object that is passed in is not a `Department` because it lacks a `code` method).

3 Graal, Truffle, Self-Optimization and Dynamic Adaptive Compilation

This section gives a brief introduction into just-in-time compilation, and the main techniques we rely on for our optimizations.

3.1 Self-Optimizing Interpreters

Self-optimizing abstract-syntax-tree (AST) interpreters [68] are the foundation for the work presented here. Together with partial evaluation [66], self-optimization enables efficient dynamic language implementations that reach the performance of custom state-of-the-art virtual machines (cf. Section 5.2 and [41]). The framework for building such interpreters is called Truffle.

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.

As an example, consider the addition of two numbers in a dynamic language, possibly written simply as: `a + b`. Because there are no static types known, the run-time values for `a` and `b` could potentially be anything from an integer or a double, to a string or a collection, or any arbitrary objects that have a "`+`" method. In an self-optimizing interpreter, the expression may be represented by an `add` node, with two child nodes that each read a variable. The first time the `add` node executes, it may find that both values to be added are integers. It will then speculate that all future executions also see integers, and thus, rewrite itself to an `add-integer` node. This `add-integer` node will simply confirm that both values are integers, and then directly perform the integer addition. Compared to a general `add` node, we do not have to cover the cases for doubles, strings, and other kinds of objects, which results in much simpler code that can be more easily optimized. All other cases are supported by rewriting the `add` node to more general versions. This happens for instance, when the values are not integers, however, programs are often very monomorphic in practice, and so the speculation is highly beneficial.

As a consequence of the rewriting, what often starts out as something close to a traditional AST, in the end incorporates additional knowledge about the execution. Thus, such trees should be referred to more correctly as *execution trees* rather than ASTs.

3.2 Polymorphic Inline Caches for Optimizing Dynamic Behavior

Polymorphic inline caches (PICs) [32] are a variation on the theme of caching run-time values to improve performance. Originally, they focused on method invocation in dynamic languages to avoid costly method lookups by caching the looked-up method for a specific type. For dynamic languages, PICs can be generalized to not only consider the receiver type, but instead for instance the object shape (cf. Section 3.3), which enables the optimizations we are aiming for.

In a language such as JavaScript, a PIC would be used for instance for the following expression: `obj.toString()`. The dot can be thought of as the lexical representation of the method lookup. An implementation would keep a small cache for each such dot in the code. This means, for each lexical lookup location, we have a separate cache. PICs benefit from the relatively monomorphic behavior of programs. A specific lexical lookup is likely to see only one kind of object in the `obj` variable, so the cache will usually have the correct method for the object ready and can avoid a costly lookup.

3.3 Object Shapes: Metadata for Dynamic Objects

Object shapes [65], which are also know as maps [20] or hidden classes, are in the most general case a type and usage profile for groups of objects. In languages such as Self, JavaScript, and Grace, we do not have traditional classes that define the set of fields for an object. The set of fields might even change over time. Furthermore, fields can theoretically store any possible value. However, in practice, the behavior of programs is again relatively monomorphic and objects created in a specific part of a program are likely to have always the same set of fields, which each are used to store only a small number of different kinds of values. For example, an object representing a counter would have a field `count`, which always stores integers, while an object representing a person may have always a field `name` that stores a string, but never an integer.

Object shapes represent this run-time information in a way that allows a just-in-time compiler to represent objects in memory similarly to C structs, and then to generate highly efficient code. Object shapes can be conflated with additional information, for instance to represent knowledge about types [6, 45]. For the use of PICs, object shapes are important, because they give objects a form of identification that groups them, and which in practice, has similar properties with respect to monomorphic behavior as classes have.

3.4 Just-in-Time Compilation with Graal and Truffle

The Graal compiler is a just-in-time compiler for Java. For languages built on the Truffle framework, Graal comes with additional support for partial evaluation, which enables efficient native code generation for Truffle interpreters [66].

As such, Graal is a metacompiler. This means that instead of compiling a specific program, in our case a Grace program, Graal compiles our Grace interpreter Moth for the execution of a specific Grace method. For simplicity, partial evaluation can be thought of a highly aggressive inlining strategy. It starts with the root node of a specific Grace method and inlines all code reachable from it, while considering the execution tree to be constant. To enable further optimizations, Graal does further inlining on the level of the Grace program, which is important to expose the same optimization opportunities classic just-in-time compilers have. The applied optimizations include for instance constant folding, common subexpression elimination, and loop-invariant code motion.

Especially loop-invariant code motion and common subexpression elimination are important to generate efficient native code for dynamic languages. Since we rely on techniques such as self-optimizing nodes, PICs and object shapes, which all introduce various checks, a compiler needs to move these out of loops, and remove redundant checks.

By combining all the techniques sketched in this section, Graal and Truffle are able to execute dynamic languages as efficiently as virtual machines built for a specific language – but with much less implementation effort.

4 Moth: Grace on Graal and Truffle

Implementing dynamic languages as state-of-the-art virtual machines can require enormous engineering efforts. Meta-compilation approaches [41] such as RPython [12, 14] and

GraalVM [67, 66] reduce the necessary work dramatically, because they allow language implementers to leverage existing VMs and their support for just-in-time compilation and garbage collection.

Moth [47] adapts SOMns [39] to leverage this infrastructure for Grace. SOMns is a Newspeak implementation [18] on top of the Truffle framework and the Graal just-in-time compiler, which are part of the GraalVM project. One key optimization of SOMns for this work is the use of object shapes [65], also known as maps [20] or hidden classes. They represent the structure of an object and the types of its fields. In SOMns, shapes correspond to the class of an object and augment it with run-time type information. With Moth's implementation, SOMns was changed to parse Grace code, adapting a few of the self-optimizing abstract-syntax-tree nodes to conform to Grace's semantics. Despite these changes Moth preserves the peak performance of SOMns, which reaches that of V8, Google's JavaScript implementation (cf. Section 5.2 and Marr *et al.* [40]).

4.1 Adding Gradual Type Checking

One of the goals for our approach to gradual typing was to keep the necessary changes to an existing implementation small, while enabling optimization in highly efficient language runtimes. In an AST interpreter, we can implement this approach by attaching the checks to the relevant AST nodes: the expected types for the argument and return values can be included with the node for requesting a method, and the expected type for a variable can be attached to the nodes for reading from and writing to that variable. In practice, we encapsulate the logic of the check within a new class of AST nodes, specially to support gradual type checking. Moth's front end was adapted to parse and record type annotations and attach instances of this checking node as children of the existing method, variable read, and variable write nodes.

The check node uses the internal representation of a Grace type (cf. Listing 4, Line 13) to test whether an observed object conforms to that type. An object satisfies a type if all members required by the type are provided by that object (Line 5).

Note, we use a pseudo code syntax similar to Python for all code examples that represent the implementation of Moth. We chose this syntax to avoid any confusion with our Grace examples (even though Moth is implemented in Java).

```
1  class Type:
2    def init(members):
3      self._members = members
4
5    def is_satisfied_by(other: Type):
6      for m in self._members:
7        if m not in other._members:
8          return False
9      return True
10
11   def check(obj: Object):
12     t = obj.get_type()
13     return self.is_satisfied_by(t)
```

Listing 4 Sketch of a `Type` in our system and its `check()` semantics.

```
1  global record: Matrix
2
3  class TypeCheckNode(Node):
4
5    expected: Type
6
7    @Spec(static_guard=`expected.check(obj)`)
8    def check(obj: Number):
9      pass
10
11   @Spec(static_guard=`expected.check(obj)`)
12   def check(obj: String):
13     pass
14
15   ...
16
17   @Spec(guard=`obj.shape==cached_shape`, static_guard=`expected.check(obj)`)
18   def check(obj: Object, @Cached(obj.shape) cached_shape: Shape):
19     pass
20
21   @Fallback
22   def check(obj: Any):
23     T = obj.get_type()
24
25     if record[T, expected] is unknown:
26       record[T, expected] = T.is_subtype_of(expected)
27
28     if not record[T, expected]:
29       raise TypeError("{obj} doesn't implement {expected}")
```

Listing 5 A sketch of the specializations in `TypeCheckNode` to minimize the run-time overhead of type checking. A specialization is a minimal set of operations for one specific situation, e.g., that the value to be checked is some type of number.

4.2 Optimization

There are two aspects to our implementation that are critical for a minimal-overhead solution:

- specialized executions of the type checking node, along with guards to protect these specialized versions, and
- a matrix to cache sub-typing relationships to eliminate redundant exhaustive subtype tests.

Optimized Type Check Node The first performance-critical aspect to our implementation is the optimization of the type checking node. We rely on Truffle and its TruffleDSL [31]. This means we provide a number of special cases, which are selected during execution based on the observed concrete kinds of objects. A sketch of our type checking node using a pseudo-code version of the DSL is given in Listing 5. A simple optimization is for well known types such as numbers (Line 8) or strings (Line 12). The methods annotated with `@Spec` (shorthand for `@Specialization`) correspond to possible states in a state machine that is generated by the TruffleDSL. Thus, if a check node observes a number or a string, it will check on the first execution only that the expected type, i.e., the one defined by some type annotation, is satisfied by the object by using a `static_guard`. If this is the case, the DSL will activate this state. For just-in-time compilation, only the activated states and their normal guards

```
1 class VariableReadNode(Node):
2   slot: FrameSlot
3   type_check: TypeCheckNode
4
5   @Spec
6   def do_read(frame: VirtualFrame):
7     value = frame.read(slot)
8     if type_check:
9       type_check.check(value)
10    return value
```

Listing 6 Sketch of a `VariableReadNode` using the `TypeCheckNode` to ensure Grace's transient semantics.

are considered. A `static_guard` is not included in the optimized code. If a check fails, or no specialization matches, the fallback, i.e., `check_generic` is selected (Line 22), which may raise a type error.

For generic objects, we rely on the specialization on Line 18, which checks that the object satisfies the expected type. If that is the case, it reads the shape of the object (cf. Section 4) at specialization time, and caches it for later comparisons. Thus, during normal execution, we only need to read the shape of the object and then compare it to the cached shape with a simple reference comparison. If the shapes are the same, we can assume the type check passed successfully. Note that shapes are not equivalent to types, however, shapes imply the set of members of an object, and thus, do imply whether an object fulfills one of our structural types.

The `TypeCheckNode` is used in Moth in all places that need to check types, which includes reading and writing variables as well as method requests and returns. Listing 6 shows a sketch of an AST node that implements reading from a local variable, which is stored in a frame object. A frame corresponds to a stack frame, sometimes also called an environment.

Line 8 first checks whether a type check needs to be performed. Since type annotations are optional, it may not be necessary to check for a type. Note that `type_check` is a constant for just-in-time compilation (cf. Section 3.4), which enables subsequent optimizations. Line 9 then calls the `check()` method on the `TypeCheckNode`, which may result in a type error. For a variable that only contains numbers, the `type_check` object would activate the number specialization in its state machine. For just-in-time compilation, this means only the code for checking numbers needs to be compiled, but none of the other specializations.

In many cases, the specialization for objects would be activated in a `TypeCheckNode`, which checks the shape of an object against a cache. This check is identical to the check performed by a polymorphic inline cache (PIC, cf. Section 3.2). Since PICs are used for all method calls, they are very common in most programs, and these additional checks can often be removed easily via common subexpression elimination.

Subtype Cache Matrix The other performance-critical aspect to our implementation is the use of a matrix to cache sub-typing relationships. The matrix compares types against types, featuring all known types along the columns and the same types again along the rows. A cell in the table corresponds to a sub-typing relationship: does the type corresponding to the row implement the type corresponding to the column? All cells in the matrix begin as unknown and, as encountered in checks during execution, we populate the table. If a particular relationship has been computed before we can skip the check and instead recall the previously-computed value (Line 26 in Listing 5). Using this table we are able to eliminate the redundancy of evaluating the same type to type relationships across different checks in the program. To reduce redundancy further we also unify types in a similar way to Java's string interning; during the construction of a type we first check to see if the same set of members is expressed by a previously-created type and, if so, we avoid creating the new instance and provide the existing one instead.

Together the self-specializing type check node and the cache matrix ensure that our implementation eliminates redundancy, and consequently, we are able to minimize the run-time overhead of our system.

5 Evaluation

To evaluate our approach to gradual type checking, we first establish the baseline performance of Moth compared to Java and JavaScript, and then assess the impact of the type checks themselves.

5.1 Method and Setup

To account for the complex warmup behavior of modern systems [4] as well as the non-determinism caused by e.g. garbage collection and cache effects, we run each benchmark for 1000 iterations in the same VM invocation.[2] Afterwards, we inspected the run-time plots over the iterations and manually determined a cutoff of 350 iterations for warmup, i.e., we discard iterations with signs of compilation. As a result, we use a large number of data points to compute the average, but outliers, caused by e.g. garbage collection, remain visible in the plots. All reported averages use the geometric mean since they aggregate ratios.

All experiments were executed on a machine running Ubuntu Linux 16.04.4, with Kernel 3.13. The machine has two Intel Xeon E5-2620 v3 2.40GHz, with 6 cores each, for a total of 24 hyperthreads. We used ReBench 0.10.1 [38], Java 1.8.0_171, Graal 0.33 (`a13b888`), Node.js 10.4, and Higgs from 9 May 2018 (`aa95240`). Benchmarks were executed one by one to avoid interference between them. The analysis of the results was done with R 3.4.1, and plots are generated with ggplot 2.2.1 and tikzDevice 0.11. Our experimental setup is available online to enable reproductions.[3]

5.2 Are We Fast Yet?

To establish the performance of Moth, we compare it to Java and JavaScript. Moth is used in its untyped version, i.e., without type checks. For JavaScript we chose two implementations, Node.js with V8 as well as the Higgs VM. The Higgs VM is an interesting point of comparison, because Richards *et al.* [45] used it in their study. The goal of this comparison is to determine whether our approach could be applicable to industrial strength language implementations without adverse effects on their performance.

We compare across languages based on the Are We Fast Yet benchmarks [40], which are designed to enable a comparison of the effectiveness of compilers across different languages. To this end, they use only a common set of core language elements. While this reduces the performance-relevant differences between languages, the set of core language elements covers

[2] For the Higgs VM, we only use 100 iterations, because of its lower performance. This is sufficient since Higgs's compilation approach induces less variation and leads to more stable measurements.

[3] SM TODO *merge changes, and tag final version* `https://github.com/gracelang/moth-benchmarks`

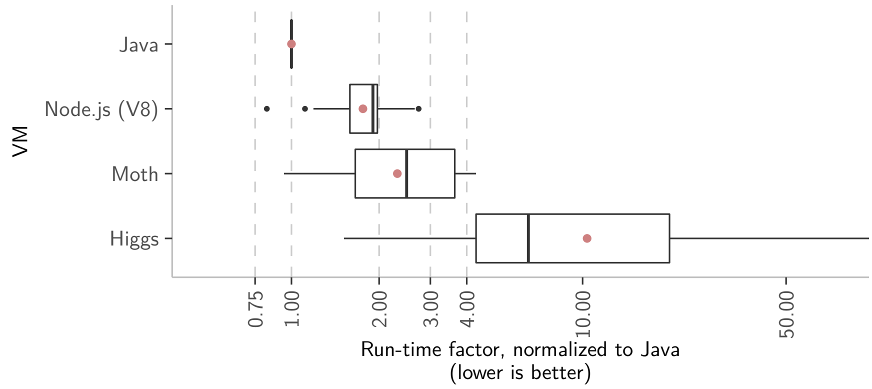


Figure 1 Comparison of Java 1.8, Node.js 10.4, Higgs VM, and Moth. The boxplot depicts the peak-performance results for the Are We Fast Yet benchmarks, each benchmark normalized individually based on the result for Java, which means all results for Java are 1.0, and its box appears as a line. The dots on the plot represent the geometric mean reported as averages. For these benchmarks, Moth is within the performance range of JavaScript, as implemented by Node.js, which makes Moth an acceptable platform for our experiments.

only common object-oriented language features with first-class functions. Consequently, these benchmarks are not necessarily a predictor for application performance, but can give a good indication for basic mechanisms such as type checking.

Figure 1 shows the results. We use Java as baseline since it is the fastest language implementation in this experiment. Note that we perform a unit conversion on the results separately for each benchmark, using the average of Java as 1 unit. While this conversion does not change the distribution of the data, it allows us to show it neatly on one plot.

We see that Node.js (V8) is about 1.8x (min. 0.8x, max. 2.7x) slower than Java. Moth is about 2.3x (min. 0.9x, max. 4.3x) slower than Java. As such, it is on average 31% (min. −16%, max. 2.3x) slower than Node.js. Compared to the Higgs VM, which is on these benchmarks 10.4x (min. 1.5x, max. 163x) slower than Java, Moth reaches the performance of Node.js more closely. With these results, we argue that Moth is a suitable platform to assess the impact of our approach to gradual type checking, because its performance is close enough to state-of-the-art VMs, and run-time overhead is not hidden by slow baseline performance.

5.3 Performance of Transient Gradual Type Checks

The performance overhead of our transient gradual type checking system is assessed based on the Are We Fast Yet benchmarks as well as benchmarks from the gradual-typing literature. The goal was to complement our benchmarks with additional ones that are used for similar experiments and can be ported to Grace. To this end, we surveyed a number of papers [56, 62, 42, 6, 45, 55, 29] and selected benchmarks that have been used by multiple papers. Some of these benchmarks overlapped with the Are We Fast Yet suite, or were available in different versions. While not always behaviorally equivalent, we chose the Are We Fast Yet versions since we already used them to establish the performance baseline. The selected benchmarks as well as the papers in which they were used are shown in Table 1.

The benchmarks were modified to have complete type information. To ensure correctness and completeness of these experiments, we added an additional check to Moth that reports absent type information to ensure each benchmark is completely typed. To assess the performance overhead of type checking, we compare the execution of Moth with all checks

Table 1 Benchmarks selected from literature.

Fannkuch	[62, 29]	
Float	[62, 42, 29]	
Go	[62, 42, 29]	
NBody	[36, 62, 29]	used [40]
Queens	[62, 42, 29]	used [40]
PyStone	[62, 42, 29]	
Sieve	[56, 42, 6, 45, 30]	used [40]
Snake	[56, 42, 6, 45, 30]	
SpectralNorm	[62, 42, 29]	

disabled, i.e., the baseline version from Section 5.2, against an execution that has all checks enabled. We did not measure programs that mix typed and untyped code because with our implementation technique a fully typed program is expected to have the largest overhead.

Peak Performance

Figure 2 depicts the overall results comparing Moth, with all optimizations, against the untyped version. The run-time overhead, after discarding the warmup iterations, is on average 5% (min. −13%, max. 79%).

The benchmark with the highest overhead of 79% is List. The benchmark traverses a linked list and has to check the list elements individually. Unfortunately, the structure of this list introduces checks that do not coincide with shape checks on the relevant objects. We consider this benchmark a pathological case and discuss it in detail in Section 6.1.

Beside List, the highest overheads are on Richards (33%), CD (12%), Snake (14%), and Towers (12%). Richards has one major component, also a linked list traversal, similar to List. Snake and Towers primarily access arrays in a way that introduces checks that do not coincide with behavior in the unchecked version.

In some benchmarks, however, the run time decreased; notably Permute (−13%), GraphSearch (−3%), and Storage (−8%). Permute simply creates the permutations of an array. GraphSearch implements a page rank algorithm and thus is primarily graph traversal. Storage stresses the garbage collector by constructing a tree of arrays. For these benchmarks the introduced checks seem to coincide with shape-check operations already performed in the untyped version. The performance improvement is possibly caused by having checks earlier, which enables the compiler to more aggressively move them out of loops. Another reason could simply be that the extra checks shift the boundaries of compilation units. In such cases, checks might not be eliminated completely, but the shifted boundary between compilation units might mean that the generated native code interacts better with the instruction cache of the processor.

Warmup Performance

To more precisely measure warmup, all relevant experiments were executed 30 times, with each running for 100 iterations. The resulting Figure 3 shows the first 100 iterations for each benchmark. For each iteration n, we normalized the measurements to the mean of iteration n of the untyped Moth implementation. Thus, any increase indicates a slow down because of

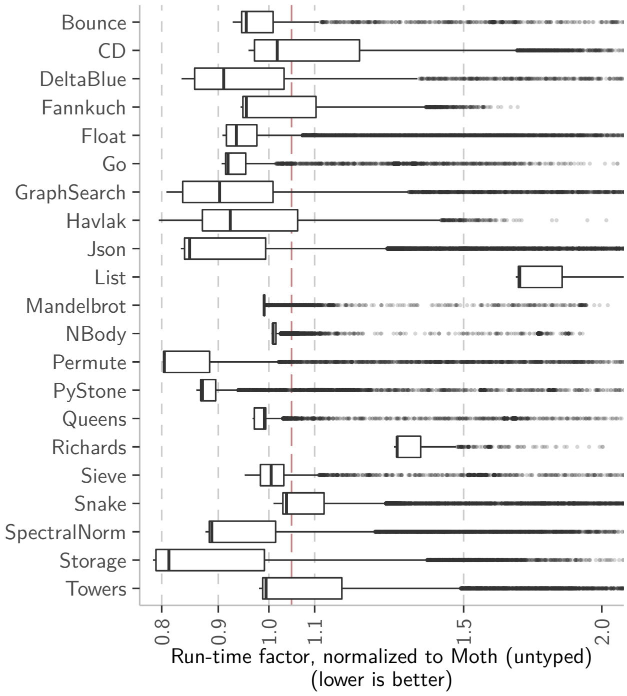


Figure 2 A boxplot comparing the performance of Moth with and without type checking. The plot depicts the run-time overhead on peak performance over the untyped performance. On average, transient type checking introduces an overhead of 5% (min. −13%, max. 79%). The average is indicated as a line with long dashes. The visible outliers correspond to various complex aspects of the overall system, e.g., including garbage collection and cache effects. Note that the axis is logarithmic to avoid distorting the proportions of relative speedups and slowdowns.

typing. The darker lines indicate the means, while the lighter area indicates a 95% confidence interval.

Looking only at the first few iterations, where we assume that most code is executed in the interpreter and might be affected by compilation activity, the overhead appears minimal. Only the Mandelbrot and CD benchmarks shows a noticeable slowdown.

Mandelbrot with its distinctly slow first iteration can be explained by its code structure. Since it is a computational kernel with many primitive operations, but no method calls, optimized code is only reached after the first full benchmark iteration. The problem could be alleviated with on-stack-replacement for loops, which is currently not done. Since other benchmarks use methods, they reach compiled code earlier and do not exhibit the same first-iteration slowdown.

PyStone however show various spikes. Since spikes appear in both directions (speedups and slowdowns), we assume that they indicate a shift, for instance, of garbage collection pauses, which may happen because of different heap configurations triggered by the additional data structures for type information.

5.4 Effectiveness of Optimizations

To characterize the concrete impact of our two optimizations, i.e., the optimized type checking node, which replaces complex type tests with checks for object shapes, and our matrix to cache sub-typing information, we look at the number of type checks performed by the

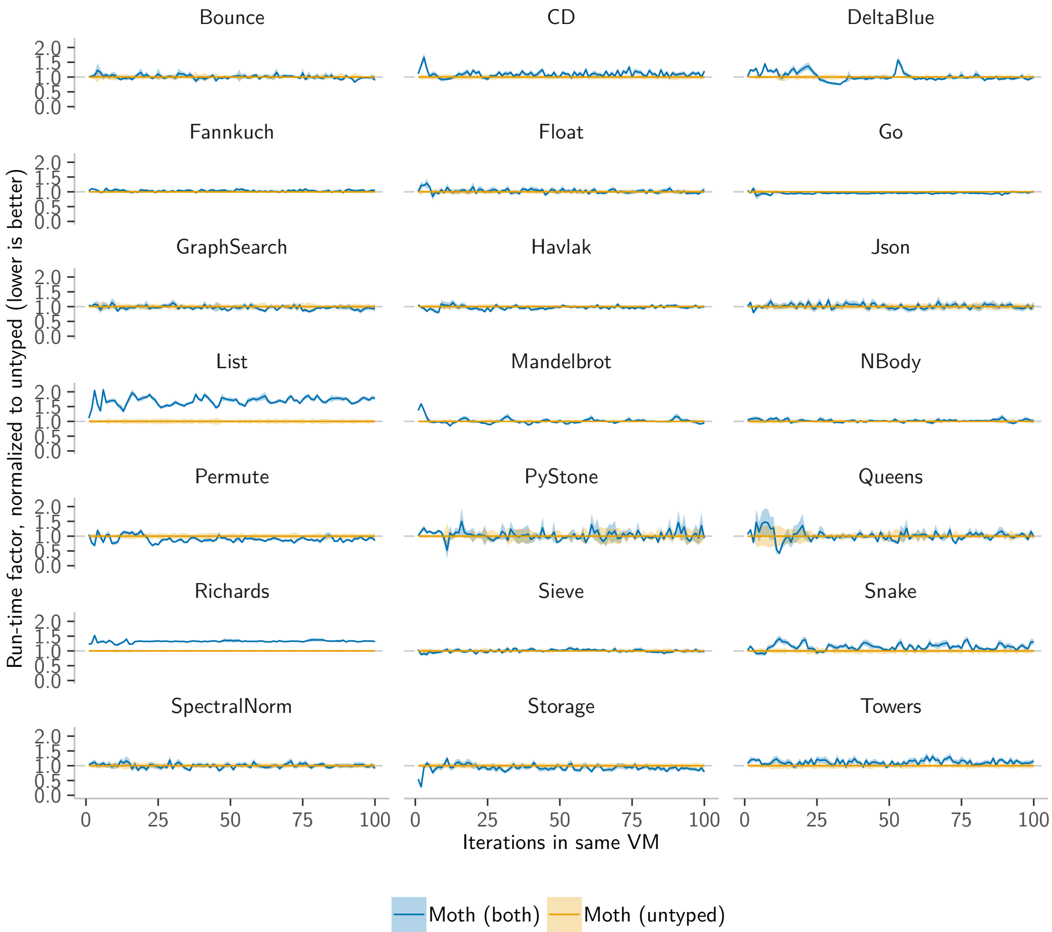


Figure 3 Plot of the run time for the first 100 iterations. The lines indicate the mean at iteration n normalized to the untyped result, the lighter area indicates a 95% confidence interval. The first iteration, i.e., mostly interpreted, seems to be affected significantly only for Mandelbrot, though CD shows slower behavior in early warmup, too.

benchmarks, as well as the impact on peak performance.

Impact on Performed Type Tests

Table 2 gives an overview of the number of type tests done by the benchmarks during execution. We distinguish two operations `check_generic` and `is_subtype_of`, which correspond to the operations in Line 22 and Line 5 of Listing 4. Thus, `check_generic` is the test called whenever a full type check has to be performed, and `is_subtype_of` is the part of the check that determines the relationship between two types. The second column of Table 2 indicates which optimization is applied, and the following columns show the mean, minimum, and maximum number of invocations of the tests over all benchmarks.

The baselines without optimizations are the rows with the results for neither of the optimizations being enabled. Depending on the benchmark, we see that the type tests are done tens of millions to hundreds of millions times for a single iteration of a benchmark.

Table 2 Type Test Statistics over all Benchmarks. This table shows how many of the type tests can be avoided based on our two optimizations. As indicated by the numbers, the number of type tests can vary significantly between benchmarks. Thus, the table shows the mean, minimum, and maximum number of type tests across all benchmarks for a given configuration. With the use of an optimized node that replaces type checks with simple object shape checks, `check_generic` is invoked only for the first time that a lexical location sees a specific object shape, which eliminates run-time type checks almost completely. Using our subtype matrix that caches type-check results, invocations of `is_subtype_of` are further reduced by an order of magnitude.

Type Test	Enabled Optimization	mean #invocations	min	max
check_generic	Neither	137,525,845	11,628,068	896,604,537
	Subtype Cache	137,525,845	11,628,068	896,604,537
	Optimized Node	292	68	1,012
	Both	292	68	1,012
is_subtype_of	Neither	134,125,215	11,628,067	896,604,534
	Subtype Cache	16	10	29
	Optimized Node	292	68	1,012
	Both	16	10	29

Our optimizations reduce the number of type test invocations dramatically. As a result, the full check for the subtyping relationship is done only once for any specific type and a possible super type. Similarly, the generic type check is replaced by a shape check and thus minimizes the number of expensive type checks to the number of lexical locations that verify types combined with the number of shapes a specific lexical location sees at run time.

Impact on Performance

Figure 4 shows how our optimizations contribute to the peak performance. The figure depicts Moth's peak performance over all benchmarks, depending on the activated optimizations. As for Figure 1, we do a per-benchmark unit conversion using Moth (untyped), preserving the distribution properties of the results, but enabling us to show the results on a single plot.

As seen before in Figure 2, the untyped version is faster by 5%. Moth with both optimizations enabled as well as Moth with the optimized type-check node (cf. Listing 4) reach the same performance. This indicates that the subtype cache matrix is not strictly necessary for the peak performance. However, we can see that the subtype cache matrix improves performance by an order of magnitude over the Moth version without any type check optimizations. This shows that it is a relevant and useful optimization. Based on the numbers of Table 2, we see that this optimization is relevant for the very first execution of code. For code that has not executed before, having the global cache for the subtype relations gives the most benefit. After the first execution, the lexical caches in form of the type check nodes are primed with the same information, and the subtype cache matrix is only rarely needed. An example for code that benefits from the subtype cache matrix is unit test code, because most of the code is executed only once. While the performance of unit tests is not always critical, it can have a major impact on developer productivity.

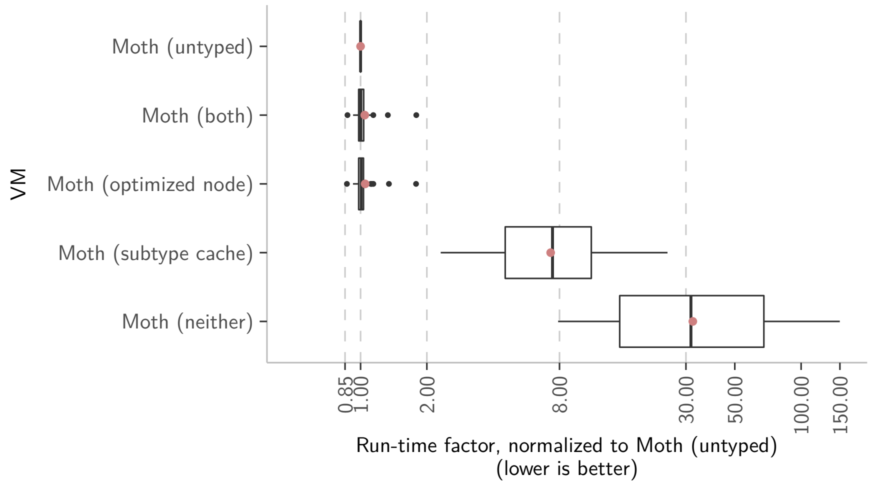


Figure 4 Performance Impact of the Optimizations on the Peak Performance over all Benchmarks. The boxplot shows the performance of Moth normalized to the untyped version, i.e., without any type checks. This means all results for Moth (untyped) are 1.0 and its box appears as a line. The dots on the plot represent the geometric mean reported as averages. The performance of Moth with both optimizations and Moth with only the node for optimized type checks are identical. The subtype check cache improves performance over the unoptimized version, but does not contribute to the peak performance.

Impact on Memory Usage

In our implementation, the subtype cache matrix is the largest additional data structure. We initialize it for up to 1000 types and use 1 byte per type combination. Java utilizes ca. 1MB of memory for the matrix. Additional memory is used to represent the type-check nodes at every lexical location. Since they behave like polymorphic inline caches (PIC) [32], their memory usage depends on the specific program execution. For the benchmarks used in this paper, the extra memory use can be up to 200KB.

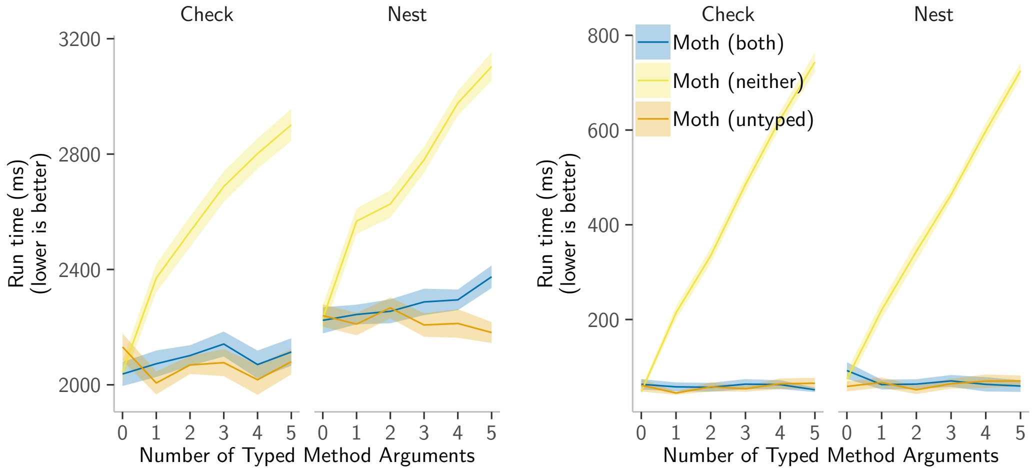
In the context of Graal and Truffle, this additional memory usage is small, since the metacompilation approach uses a lot of memory [41]. In a dedicated virtual machine, memory use can be further optimized and be as efficient as for other kinds of PICs.

5.5 Transient Typechecks are (Almost) Free

As discussed in the introduction, in many existing gradually typed systems, one would expect a linear increase of the performance overhead with the increasing number of type annotations.

In this section, we show that this is not necessarily the case on our system. For this purpose, we use two microbenchmarks Check and Nest, which have at their core method calls with 5 parameters. The Check benchmark calls the same method 10 times in a row, i.e., it has 10 call sites. The Nest benchmark has 10 methods with identical signatures, which recurse from the first one to the last one. Thus, there are still 10 method calls, but they are nested in each other. In both benchmarks, each method increments a counter, which is checked at the end of the execution to verify that both do the same number of method activations, and only the shape of the activation stack differs.

Each benchmark exists in six variants, each variant in a separate file, going from having no type annotations over annotating only the first method parameter to annotating all 5 parameters. To demonstrate the impact of compilation, we present the results for the first



(a) Iteration 1.

(b) Iteration 100.

Figure 5 Transient Typechecks are (Almost) Free. Two microbenchmarks, each with six variants, demonstrate the common scenario of adding type annotations over time, which in our system does not have an impact on peak performance. The benchmark variants differ only in the increasing number of method arguments that have type annotations. We show the result for the first benchmark iteration (a) and the one hundredth (b). Moth (neither), i.e., Moth without our two optimizations sees a linear increase in run time. For the first iteration, we see some difference between Moth (both) and Moth (untyped). By the hundredth iteration, however, the compiler has eliminated the overhead of the type checks and both Moth variants essentially have the same performance (independent of the number of method arguments with type annotations).

iteration as well as the hundredth iteration. The first iteration is executed at least partially in the interpreter, while the hundredth iteration executes fully compiled.

Figure 5 shows that such a common scenario of methods being gradually annotated with types does not incur an overhead on peak performance in our system. The plot shows the mean of the run time for each benchmark configuration. Furthermore, it indicates a band with the 95% confidence interval. The yellow line, Moth (neither), corresponds to our Moth with type checking but without any optimizations. For this case, we see that the performance overhead grows linearly with the number of type annotations.

For Moth (both) and Moth (untyped), we see for the first iteration that the band of confidence intervals diverges, indicating that the additional type checks have an impact on startup performance. However, for the hundredth iteration, the confidence intervals overlap for the optimized Moth as well as the one that does not perform typechecks. This means that Moth does not suffer from a general linear overhead for adding type checks. Instead, most type checks do not have an impact on peak performance. However, as previously argued for the List benchmark, this is only the case for checks that can be subsumed by shape checks (shape checks are performed whether or not type checks are present).

5.6 Changes to Moth

Outlined earlier in Section 4, a secondary goal of our design was to enable the implementation of our approach to be realized with few changes to the underlying interpreter. This helps to ensure that each Grace implementation can provide type checking in a uniform way.

By examining the history of changes maintained by our version control, we estimate that

```
type ListElement = interface {
  next
}

var elem: ListElement := headOfList
while (...) do {
  elem := elem.next
}
```

Listing 7 Example for dynamic type checks not corresponding to existing checks.

our implementation of Moth required 549 new lines and 59 changes to existing lines. The changes correspond to the implementation of new modules for the type class (179 lines) and the self-specializing type checking node (139 lines), modifications to the front end to extract typing information (115 new lines, 14 lines changes) and finally the new fields and amended constructors for AST nodes (116 new lines, 45 lines changes).

6 Discussion

6.1 The VM Could Not Already Know That

One of the key optimizations for our work and the work of others [6, 45] is the use of object shapes to encode information about types (in our case), or type casts and assumptions (in the case of gradually typed systems). The general idea is that a VM will already use object shapes for method dispatches, field accesses, and other operations on objects. Thus any further use to also imply type information can often be optimized away when the compiler sees that the same checks are done, and therefore can be combined by optimizations such as common subexpression elimination.

This assumption breaks, however, when checks are introduced that do not correspond to those that exist already. As described in Section 4, our approach introduces checks for reading from and writing to variables. Listing 7 gives an example of a pathological case. It is a loop traversing a linked list. For this example our approach introduces a check, for the `ListElement` type, when (1) assigning to and reading from `elem` and (2) when activating the `next` method. The checks for reading from `elem` and activating the method can be combined with the dispatch's check on object shape. Unfortunately, the compiler cannot remove the check when writing to `elem`, because it has no information about what value will be returned from `next`, and so it needs to preserve the check to be able to trigger an error on the assignment. For our List benchmark, this check induces an overhead of 79%.

Compiler optimizations such as inlining are also insufficient for this particular case, because there are no guarantees about what `elem` does to implement `next`. The `next` method of a specific kind of `ListElement` may even have a type annotation for a return value. The best Graal can do in this example is to combine the check for the return value with the one writing to `elem`.

Since the example shows a somewhat generic data structure, there is the question of whether the issue applies to other data structures as well. Our benchmarks use a range of data structures including hash maps, sets, and vectors, none of which show the issue, because in more complex programs the chance of already having a check there is high, and cases were there has not been one before seem to be rare — although one can always consider additional optimizations to eliminate further checks. For generic data structures, storage strategies [13] could be used to encode type information about elements. This would allow the VM to check only once before a loop, and the loop could then rely on that check for guarantees about the elements of the data structure.

6.2 Optimizations

Read and Write Checks. As a simplification, we currently check variable access on both reads and writes. This approach simplifies the implementation, because we do not need to adapt all built-ins, i.e., all primitive operations provided by the interpreter. One optimization could be to avoid read checks. A type violation can normally only occur when writing to a variable, but not when reading. However, to maintain the semantics, this would require us to adapt many primitives. Examples for primitives are operations that activate blocks, which need to check their arguments or return values as well as any primitives that write to variables or fields. Given the number of primitives, this is error prone and incompleteness would result in missing type checks.

By checking reads and writes in a few well defined locations, we get errors as soon as user code accesses fields and variables. Moreover, only a small set of locations required changes to the code, which simplified the implementation. Given the good results (cf. Sections 5.4 and 5.6), we decided to keep read checks, because it is a more uniform and maintainable approach for an academic project.

Dynamic Type Propagation. Another optimization could be to use Truffle's approach to self-specialization [68] and propagate type information to avoid redundant checks. At the moment, Truffle interpreters typically use self-specialization to specialize the AST to avoid boxing of primitive types. This is done by speculating that some subtree always returns the expected type. If this is not the case, the return value of the subtree is going to be propagated via an exception, which is caught and triggers respecialization. This idea could possibly be used to encode higher-level type information for return values, too. This could be used to remove redundant checks in the interpreter by simply discovering at run time that whole subexpressions conform to the type annotations.

Performance Impact of Types As seen in Section 6.1, there are cases where adding types may reduce performance, even so, in the best case this does not happen (cf. Section 5.5).

While the expectation is that adding more types may result in higher potential for performance issues, in the context of dynamic and adaptive compilation as used for Moth, this is not necessarily the case. Since compilers rely on various heuristics, for instance for inlining, there may be situations where a fully typed program is faster than a program with fewer types. Since the checks need to be compiled themselves, they also influence such heuristics. This means it is possible that partially typed programs may show worse performance than fully typed ones.

6.3 Threats to Validity

This work is subject to many of the threats to validity common to evaluations of experimental language implementations. Our underlying implementation may contain undetected bugs that affect the semantics or performance of the gradual typing checks, affecting construct validity — we may not have implemented what we think we have. Given that, our benchmarking harness run on the same implementation is subject to the same risks, thus also affecting internal validity — we may not be measuring the implementation correctly. Moth is built on the Truffle and Graal toolchain, so we expect external validity there at least — we expect the results would transfer to other Graal VMs doing similar AST-based optimizations. We have less external validity regarding other kinds of VMs (such as VMs specialized to particular languages, or VMs built via meta-tracing rather than partial evaluation). Nevertheless, we expect our results should be transferable as we rely on common techniques.

Generalizability Finally, because we are working in Grace, it is less obvious that our results generalize to other gradually typed-languages. We have worked to ensure e.g. our benchmarks do not depend on any features of Grace that are not common in other gradually-typed object-oriented languages, but as Grace lacks a large corpus of programs the benchmarks are necessarily artificial, and it is not clear how the results would transfer to the kinds of programs actually written in practice. The advantage of Grace (and Moth) 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. On the other hand, more effort on optimisations could well lead to even better performance.

Another aspect which limits generalizability is the specific semantics of Grace. Reticulated Python, Typed Racket, and Gradualtalk have semantics that need additional runtime support, and thus, we cannot draw any conclusions without further research.

For languages such as Newspeak, Strongtalk, or TypeScript, where types do not have run-time semantics, one could add termination based on type errors to these languages, or simply avoid termination and report the errors after program completion as a debugging aid. For either option, our approach should apply and we would expect similar results.

7 Related Work

Although syntaxes for type annotations in dynamic languages go back at least as far as Lisp [54], the first attempts at adding a comprehensive static type system to a dynamically typed language involved Smalltalk [33], with the first practical system being Bracha's Strongtalk [17]. Strongtalk (independently replicated for Ruby [26]) provided a powerful and flexible static type system, where crucially, the system was *optional* (also known as pluggable [16]). Programmers could run the static checker over their Smalltalk code (or not); either way the type annotations had no impact whatsoever of the semantics of the underlying Smalltalk program.

Siek and Taha [48] introduced the term "gradual typing" to describe the logical extension of this scheme: a dynamic language with type annotations that could, if necessary, be checked at runtime. Siek and Taha build on earlier complementary work extending fully statically typed languages with a "`DYNAMIC`" type—Abadi *et al.* 's 1991 TOPLAS paper [1] is an important early attempt and also surveys previous work.

Revived practical adoption of dynamic languages generated revived research interest, leading to the formulation of the *gradual guarantee* to characterize sound gradual type systems: informally "removing type annotations always produces a program that is still well typed" and also "evaluates to an equivalent value" [50], drawing on Boyland's critical insight that such a guarantee must by its nature exclude code that reflects on the presence or absence of type declarations [15]. Moth ensures that the values passing through type annotations cannot be incompatible with those annotations and that type annotations cannot change program values, and Moth's type tests consider only method names (not any further type annotations). This means that removing type annotations cannot cause a program to fail or change its behaviour, satisfying the informal statement of the gradual guarantee. Moth does not meet the refined formal statement of the guarantee in Siek*et al.*'s [50]'s Theorem 5, however, because Theorem 5 requires all intermediate values conform to their inferred static types. Moth only checks at explicit type declarations, not inferred intermediate types.

Type errors in gradual, or other dynamically checked, type systems will be detected at the type declarations, but often those declarations will not be at fault — indeed in a correctly typed program in a sound gradually typed system, the declarations cannot be at fault because they will have passed the static type checker. Rather, the underlying fault must be somewhere within the barbarian dynamically typed code *trans vallum.* Blame tracking [63, 52, 2] localizes these faults by identifying the point in the program where the system makes an assumption about dynamically typed objects, so can identify the root cause should the assumption fail. Different semantics for blame detect these faults slightly differently, and impose more or less implementation overhead [60, 51, 62].

The diversity of semantics and language designs incorporating gradual typing has been captured recently via surveys incorporating formal models of different design options. Chung *et al.* [22] present an object-oriented model covering optional semantics (erasure), transient semantics, concrete semantics (from Thorn [11]), and behavioural semantics (from Typed Racket), and give a series of programs to clarify the semantics of a particular language. Greenman *et al.* take a more functional approach, again modelling erasure, transient ("first order"), and behavioural ("higher order") semantics [28], and also present performance information based on Typed Racket. Wilson *et al.* take a rather different approach, employing questionnaires to investigate the semantics programmers expect of a gradual typing system [64].

As with languages more generally, there seem to be two main implementation strategies for languages mixing dynamic and static type checks: either adding static checks into a dynamic language implementation, or adding support for dynamic types to an implementation that depends on static types for efficiency. Typed Racket, for example, optimizes code with a combination of type inference and type declarations—the Racket IDE "optimizer coach" goes as far as to suggest to programmers type annotations that may improve their program's performance [53]. In these implementations, values flowing from dynamically to statically typed code must be checked at the boundary. Fully statically typed code needs no dynamic type checks, and so generally performs better than dynamically typed code. Adopting a gradual type system such as Typed Racket [59] allows programmers to explicitly declare types that can be checked statically, removing unnecessary overhead. Ortin *et al.*'s [43] approach takes this to a logical extreme using a rule base to guide program specialisation at compile time based on abstract interpretation.

On the other hand, systems such as Reticulated Python [60], SafeTypeScript [45], and our work here, take the opposite approach. These systems do not use information from type declarations to optimize execution speed, rather the necessity to perform (potentially repeated) dynamic type checks tends to slow programs down, so here code with no type annotations generally performs better than statically typed code, or rather, code with many type annotations. In the limit, these kinds of systems may only ever check types dynamically and may not involve a static type checker at all.

As gradual typing systems have come to wider attention, the question of their implementation overheads has become more prominent. Takikawa *et al.* [56] asked "is sound gradual typing dead?" based on a systematic performance measurement on Typed Racket. The key here is their evaluation method, where they constructed a number of different permutations of typed and untyped code, and evaluated performance along the spectrum [30].

Bauman *et al.* [6] replied to Takikawa *et al.*'s study, in which they used Pycket [5] (a tracing JIT for Racket) rather than the standard Racket VM, but maintained full gradually-typed Racket semantics. Bauman *et al.* are able to demonstrate most benchmarks with a slowdown of 2x on average over all configurations. Note that this is not directly comparable to our system, 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, however, they use the same essential optimization technique that we apply, using object shapes to encode information about gradual types. Muehlboeck and Tate [42] also replied to Takikawa *et al.*, using a similar benchmarking method applied to Nom, a language with features 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.

Most recently, Kuhlenschmidt *et al.* [36] employ an ahead of time (i.e. traditional, static) compiler for a custom language called Grift and demonstrate good performance for code where more than half of the program is annotated with types, and reasonable performance for code without type annotations.

Perhaps the closest to our approach are Vitousek *et al.* [60] (incl. [62, 29]) and Richards *et al.* [45]. Vitousek *et al.* describe dynamically checking transient types for Reticulated Python (termed "tag-type" soundness by Greenman and Migeed [29]). As with our work, Vitousek *et al.*'s transient checks inspect only the "top-level" type of an object. Reticulated Python undertakes these transient type checks at different places to Moth. Moth only checks explicit type annotations, while Reticulated Python implicitly checks whenever values flow from dynamic to static types. We refrain from a direct performance comparison since Reticulated Python is an interpreter without just-in-time compilation and thus performance tradeoffs are different. In recent experimental work, however, Vitousek *et al.* [61] have evaluated Reticulated Python's transient semantics running on top of an unmodified PyPy JIT metacompiler. These results are broadly consistent with those presented here, finding similarly small slowdowns using just the tracing JIT, and reducing those slowdowns even further when some tests are elimited via static type inference.

Richards *et al.* [45] take a similar implementation approach to our work, demonstrating that key mechanisms such as object shapes used by a VM to optimize dynamic languages can be used to eliminate most of the overhead of dynamic type checks. Unlike our work, Richards implement "monotonic" gradual typing with blame, rather than the simpler transient checks, and do so on top of an adapted Higgs VM. The Higgs VM implements a baseline just-in-time compiler based on basic-block versioning [21]. In contrast, our implementation of dynamic checks is built on top of the Truffle framework for the Graal VM, and reaches performance approaching that of V8 (cf. Section 5.2). The performance difference is of relevance here since any small constant factors introduced into a VM with a lower baseline performance can remain hidden, while they stand out more prominently on a faster baseline.

Overall, it is unclear whether our results confirm the ones reported by Richards *et al.* [45], because our system is simpler. It does not introduce the polymorphism issues caused by accumulating cast information on object shapes, which could be important for performance. Considering that Richards *et al.* report ca. 4% overhead on the classic Richards benchmark, while we see 33%, further work seems necessary to understand the performance implications of their approach for a highly optimizing just-in-time compiler.

8 Conclusion

As gradually typed languages become more common, and both static and dynamically typed languages are extended with gradual features, efficient techniques for gradual type checking become more important. In this paper, we have demonstrated that optimizing virtual machines enable transient gradual type checks with relatively little overhead, and with only small modifications to an AST interpreter. We evaluated this approach with Moth, an implementation of the Grace language on top of Truffle and Graal.

In our implementation, types are structural and shallow: a type specifies only the names of members provided by objects, and not the types of their arguments and results. These types are checked on access to variables, when assigning to method parameters, and also on return values. The information on types is encoded as part of an object's shape, which means that shape checks already performed in an optimizing dynamic language implementation can be used to check types, too. Being able to tie checks to the shapes in this way is critical for reducing the overhead of dynamic checking.

Using the Are We Fast Yet benchmarks as well as a collection of benchmarks from the gradual typing literature, we find that our approach to dynamic type checking introduces an overhead of 5% (min. −13%, max. 79%) on peak performance. In addition to the results from further microbenchmarks, we take this as a strong indication that transient gradual types do not need to imply a linear overhead compared to untyped programs. However, we also see that interpreter and startup performance is indeed reduced by additional type annotations.

Since Moth reaches the performance of a highly optimized JavaScript VM such as V8, we believe that these results are a good indication for the low peak-performance overhead of our approach.

In specific cases, the overhead is still significant and requires further research to be practical. Thus, future research should investigate how the number of gradual type checks can be reduced without causing the type feedback to become too imprecise to be useful. One approach might increase the necessary changes to a language implementation, but avoid checking every variable read. Another approach might further leverage Truffle's self-specialization to propagate type requirements and avoid unnecessary checks.

Finally, we hope to apply our approach to other parts of the spectrum of gradual typing, eventually reaching full structural types with blame that support the gradual guarantee. This should let us verify that Richards *et al.* [45]'s results generalize to highly optimizing virtual machines, or alternatively, show that other optimizations for precise blame need to be investigated.

References

1 Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

2 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 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*, pages 201–214, 2011.

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

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

5 Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 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*, pages 22–34, 2015.

6 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA):54:1–54:24, October 2017.

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

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

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

10 Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. 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*, pages 1–51, 2007.

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

12 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25. ACM, 2009.

13 Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'13, pages 167–182. ACM, 2013.

14 Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 98:408–424, February 2013.

15 John Tang Boyland. The problem of structural type tests in a gradual-typed language. In *FOOL*, 2014.

16 Gilad Bracha. Pluggable Type Systems. OOPSLA Workshop on Revival of Dynamic Languages, October 2004.

17 Gilad Bracha and David Griswold. Stongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, 1993.

18 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *European Conference on Object-*

Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010.

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

20 Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA'89, pages 49–70. ACM, October 1989.

21 Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *LIPIcs*, pages 7:1–7:24. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.7.

22 Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. KafKa: gradual typing for objects. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 12:1–12:24, 2018. doi:10.4230/LIPIcs.ECOOP.2018.12.

23 Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 105–117. ACM, 2015.

24 Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 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, pages 642–659. ACM, 2016.

25 Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. Idle Time Garbage Collection Scheduling. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'16, pages 570–583. ACM, 2016.

26 M. Furr, J.-H. An, J. Foster, and M.J. Hicks. Static type inference for Ruby. In *Symposium on Applied Computation*, pages 1859–1866, 2009.

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

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

29 Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM'18, pages 30–39. ACM, 2018.

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

31 Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 13th International Conference on Generative Programming: Concepts and Experiences*, GPCE '14, pages 123–132. ACM, 2014. doi:10.1145/2658761.2658776.

32 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Con-*

ference on Object-Oriented Programming, volume 512 of *LNCS*, pages 21–38. Springer, 1991. doi:10.1007/BFb0057013.

33 Ralph E. Johnson. Type-checking Smalltalk. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings.*, pages 315–321, 1986.

34 Timothy Jones. *Classless Object Semantics*. PhD thesis, Victoria University of Wellington, 2017.

35 Timothy Jones, Michael Homer, James Noble, and Kim Bruce. Object inheritance without classes. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56, pages 13:1–13:26, 2016.

36 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Efficient Gradual Typing. *CoRR*, abs/1802.06375, 2018. arXiv:1802.06375.

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

38 Stefan Marr. Rebench: Execute and document benchmarks reproducibly, July 2018. Version 0.10.1. doi:10.5281/zenodo.1311762.

39 Stefan Marr. SOMns: a newspeak for concurrency research. https://github.com/-smarr/SOMns, 2018.

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

41 Stefan Marr and Stéphane Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '15, pages 821–839. ACM, October 2015.

42 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proc. ACM Program. Lang.*, 1(OOPSLA):56:1–56:30, October 2017.

43 Francisco Ortin, Miguel Garcia, and Seán McSweeney. Rule-based program specialization to optimize gradually typed code. *Knowledge-Based Systems*, 2019. doi:https://doi.org/10.1016/j.knosys.2019.05.013.

44 Aaron Pang, Craig Anslow, and James Noble. What programming languages do developers use? A theory of static vs dynamic language choice. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2018, Lisbon, Portugal, October 1-4, 2018*, pages 239–247, 2018. doi:10.1109/VLHCC.2018.8506534.

45 Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM already knew that: Leveraging compile-time knowledge to optimize gradual typing. *Proc. ACM Program. Lang.*, 1(OOPSLA):55:1–55:27, October 2017.

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

47 Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 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*, 2017. Presentation.

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

49 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, pages 2–27, 2007.

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

51 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *European Symposium on Programming (ESOP)*, pages 432–456, 2015.

52 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 365–376, 2010.

53 Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. 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*, pages 163–178, 2012.

54 G.L. Steele. *Common Lisp the Language.* Digital Press, 1990.

55 Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. Reducing the overhead of assertion run-time checks via static analysis. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP'16, pages 90–103. ACM, 2016.

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

57 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 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*, pages 793–810, 2012. `doi:10.1145/2384616.2384674`.

58 The Clean. Vehicle. Flying Nun Records, FN147, 1990.

59 Sam Tobin-Hochstadt and Matthias Felleisen. 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*, pages 395–406, 2008.

60 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 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*, pages 45–56, 2014.

61 Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and evaluating transient gradual typing. *CoRR*, abs/1902.07808, 2019. URL: `http://arxiv.org/abs/1902.07808`, `arXiv:1902.07808`.

62 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems.

In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL'17, pages 762–774. ACM, 2017.

63 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming Languages and Systems (ESOP)*, pages 1–16, 2009.

64 Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The behavior of gradual types: A user study. In *Dynamic Language Symposium (DLS)*, 2018.

65 Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 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, pages 133–144. ACM, 2014.

66 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 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, pages 662–676. ACM, 2017.

67 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 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, pages 187–204. ACM, 2013.

68 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium*, DLS'12, pages 73–82, October 2012. `doi:10.1145/2384577.2384587`.