Is Sound Gradual Typing Dead?

Dr. Double B. Reviewing, I
In Famous University
turing@award.com

Dr. Double B. Reviewing, II
Less Famous University
turing@award.com

Dr. Double B. Reviewing, III
In Famous University
turing@award.com

Dr. Double B. Reviewing, IV
In Famous University
turing@award.com

Dr. Double B. Reviewing, V
Very Famous University
turing@award.com

Dr. Double B. Reviewing, VI
Somewhat Famous University
turing@award.com

Abstract

Programmers have come to embrace dynamically-typed languages for prototyping and delivering large and complex systems. When it comes to maintaining and evolving these systems, the lack of explicit static typing becomes a bottleneck. In response, researchers have explored the idea of gradually-typed programming languages which allow the post-hoc addition of type annotations to software written in one of these untyped languages. Some of these new, hybrid languages insert run-time checks at the boundary between typed and untyped code to establish type soundness for the overall system. With sound gradual typing, programmers can rely on the language implementation to provide meaningful error messages when type invariants are violated.

While most of the research on sound gradual typing has remained theoretical, the few emerging implementations suffer from performance overheads due to these checks. None of the publications on this topic come with a comprehensive performance evaluation. However, a few report disastrous numbers. In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method takes the idea of a gradual conversion from untyped to typed seriously and calls for measuring the performance of all possible partial conversions of a given untyped benchmark. The paper reports on the results of applying the method to Typed Racket, a mature implementation of sound gradual typing, and a suite of real-world programs of various sizes and complexities. Based on the results obtained in this study, the paper concludes that, given the current state of implementation technologies, sound gradual typing is dead. Conversely, it raises the question of how implementations could reduce the overheads associated with soundness and how tools could be used to steer programmers clear from pathological cases.

, .

http://dx.doi.org/10.1145/

1. Gradual Typing and Performance

Over the past couple of decades dynamically-typed languages have become a staple of the software engineering world. Programmers use these languages to build all kinds of software systems. In many cases, the systems start as innocent prototypes. Soon enough, though, they grow into complex, multi-module programs, at which point the engineers realize that they are facing a maintenance nightmare, mostly due to the lack of reliable type information.

Gradual typing [20, 25] proposes a language-based solution to this pressing software engineering problem. The idea is to extend the language so that programmers can incrementally equip programs with types. In contrast to optional type systems [9], gradual type systems provide programmers with soundness guarantees. In other words, the type annotations in gradual type systems correctly predict run-time behavior.

Realizing type soundness in this world requires run-time checks that watch out for potential impedance mismatches between the typed and untyped portions of the programs. The granularity of these checks determine the peformance overhead of gradual typing. To reduce the frequency of checks, *macro-level* gradual typing forces programmers to annotate entire modules with types and relies on behavioral contracts [11] between typed and untyped modules to enforce soundness. Micro-level gradual typing instead assigns an implicit type `Dyn` [1] to all unannotated parts of a program; type annotations can then be added to any declaration. It is up to the implementation to insert casts at the appropriate points in the code. Different language designs use slightly different semantics with different associated costs and limitations.

Both approaches to gradual typing come with two implicit claims. First, the type systems accommodate common untyped programming idioms. This allows programmers to add types with minimal changes to existing code. Second, the cost of soundness is tolerable, meaning programs remain performant even as programmers add type annotations. Ideally, types should improve performance as they provide invariants that an optimizing compiler can leverage. While almost every publication on gradual typing validates some version of the first claim, no projects tackle the second claim systematically. Most publications come with qualified remarks about the performance of partially typed programs. Some plainly admit that such mixed programs may suffer performance degradations of two orders of magnitude.

This paper introduces a method for the systematic performance evaluation of gradual type systems. The paper's insight is that to understand the performance of a gradual type system it is necessary to simulate how a maintenance programmer chooses to add types to an existing software system. For practical reasons, such as time or

access to source code, it may be possible to add types to only a part of the system. In the context of a macro-level gradual type system, all n modules are annotated with types, and the resulting collection of $2 \cdot n$ modules is then used to create all 2^n configurations. The collection of these configurations forms a complete lattice with the untyped configuration at the bottom and the completely typed one at the top. In between, the lattice contains configurations in which some modules are typed and others are untyped. Adding types to an untyped module in one of these configurations yields a configuration at the next level of the lattice. In short, the lattice mimics all possible choices of single-module type conversions a programmer faces when a maintenance task comes up.

A performance evaluation of a gradual type system must run and time every configuration for every benchmark and extract information from these timings. The timings may answer basic questions such as how many of these configurations could be deployed without degrading performance too much.

We apply our method to Typed Racket, the gradually typed sister language of Racket, on a collection of programs ranging from 150 to 7.5K lines of code. Typed Racket is the oldest (developed since 2006) and probably most sophisticated implementation of gradual typing, but it is also a natural choice because its macro-level approach to gradual typing may impose a lower cost for boundaries between typed and untyped modules compared to competing approaches. Furthermore, since Racket is a widely used programming language, Typed Racket has also rapidly acquired a fair number of users in the commercial and open source community, which suggests at least adequate performance.[1]

Section 2 introduces the evaluation method in detail, including the information we retrieve from the lattices and how we parameterize these retrievals. Next, section 3 explains our specific benchmarks. Section 4 presents the numeric results. Section 5 discusses these results. Section 6 reviews the literature on gradual type systems with respect to performance evaluation. Section 7 concludes with ideas for future work.

2. Benchmarking Software Evolution

The inspiration for our evaluation method is due to Takikawa et al. [24]. For their extension of functional Typed Racket to the object-oriented aspects of Racket, they use a lattice-style approach for their preliminary performance evaluation. By inspecting the entire lattices of typed/untyped configurations of two small game systems, they are able to identify and then eliminate a major performance bottleneck from their implementation. Takikawa et al.'s performance evaluation was conducted in tandem with the design and implementation of Typed Racket, and thus their final results are relatively positive. In contrast, we conduct our evaluation completely *independently* of Typed Racket's implementation efforts.[2]

Let us articulate Takikawa et al.'s view from our perspective:

- A (*software system*) *configuration* is a sequence of *n* modules.
- Each module in a software system configuration is either typed or untyped.
- A configuration c_t is greater than a configuration c_u (or equal) if c_t uses a typed module for every position in the sequence for which c_u uses a typed module.
- **Proposition** The collection of all configurations of length *n* forms a complete lattice of size 2^n. The bottom element is the completely untyped configuration; the top element is the completely typed one.

We speak of a *performance lattice* to describe this idea.

Our contribution is to exploit the lattice-oriented approach to benchmarking for a *summative* evaluation. To this end, we imagine software engineers who are considering the use of gradual typing for some system and consider what kinds of questions may influence their decision. Based on this first step, we formulate a small number of parameterized, quantitative measures that capture possible answers to these questions.

When the configuration consists of a small number of modules, the software engineers might be able to equip the entire system with type annotations in one fell swoop. Such a fully annotated system should perform as well as the original, untyped version—and if the gradual type system is integrated with the compiler, it may even run faster because the compiler can apply standard type-based optimization techniques.

Definition (*typed/untyped ratio*) The typed/untyped ratio of a performance lattice is the time needed to run the top configuration divided by the time needed to run the bottom configuration.

Unfortunately, this assumption overlooks the realities of implementations of gradual typing. Some modules simply cannot be equipped with types because they use linguistic constructs that cannot be annotated. Furthermore, completely typed configurations still use the run-time libraries of the underlying untyped language. In particular, Typed Racket's run-time system remains largely untyped. As a result, even the completely typed configurations of our benchmarks usually import constants, functions, and classes from an untyped module in the run-time system. When these values cross this boundary at run-time, the contract system performs checks, and that imposes additional costs. To address this issue, the implementors of Typed Racket have enlarged their trusted code base with unchecked type environments that cover frequently imported parts of the run-time system. The next section explains what "completely typed" means for the individual benchmarks.

When the software system configuration consists of a reasonably large number of modules, no software engineering team can annotate the entire system with types all at once. Every effort is bound to leave the configuration in a state where some modules are typed and some others are untyped. As a result, the configuration is likely to suffer from the software contracts that the gradual type system injects at the boundary between the typed and the untyped portions. If the cost is tolerable, the configuration can be released and can replace the currently deployed version. The projects working on macro-level approaches to gradual typing seem to assume this kind of mode of operation. The run-time costs may not be tolerable, however, as Takikawa et al. observe. In that case, the question is how much more work the software engineers have to invest to reach a releasable configuration. That is, how many more modules must be converted before the performance is good enough for the new configuration to replace the running one.

To capture this idea, we formulate the following definition of "deliverable configurations."

Definition (*N-deliverable*) A configuration in a performance lattice is *N*-deliverable if its performance is no worse than a N% slowdown compared to the completely untyped configuration.

We parameterize this definition over the slowdown percentage that a team may consider acceptable. One team may think of a 10% slowdown as barely acceptable, while another one may tolerate a slowdown of an order of magnitude [24]. The performance evaluations below show *N*-deliverable for a range of values of *N* for all of our performance lattices.

[1] Personal communication with the implementors, who claim some 100,000 unique downloads per year.

[2] In Takikawa et al.'s terminology, borrowed from the education community [19], they conducted a *formative evaluation* while we conduct a *summative evaluation* to assess the post-intervention state of the system.

Even if a configuration is not deliverable, it might be suitably fast to run the test suites and the stress tests. A software engineering team can use such a configuration for development purposes, but it may not deliver it. The question is how many configurations of a performance lattice are usable in that sense. In order to formulate this criteria properly, we introduce the following definition of usable configurations.

Definition (*N/M-usable*) A configuration in a performance lattice is *N/M*-usable if its performance is worse than a N% slowdown and no worse than an M% slowdown compared to the completely untyped configuration.

Like the preceding definition, this one is also parametrized. Using the first parameter, we exclude deliverable configurations from the count. The second parameter specifies the positive boundary, that is, the acceptable slowdown percentage for a usable configuration. The performance evaluations below report *N/M*-usable for two values of *M* per value of *N*.

Definition (*unacceptable*) For any choice of *N* and *M*, a configuration is unacceptable if it is neither *N*-deliverable nor *N/M*-usable.

Finally, we can also ask the question how much work a team has to invest to turn unacceptable configurations into useful or even deliverable configurations. In the context of macro-level gradual typing, one easy way to measure this amount of work is to count the number of modules that have to be annotated with types before the resulting configuration becomes usable or deliverable. Here is the precise definition.

Definition (*L-step N/M-usable*) A configuration is *L*-step *N/M*-usable if it is unacceptable and at most *L* type conversion steps away from a *N*-deliverable or a *N/M*-usable configuration.

Again, the performance reports below varies the *L* parameter.

This paper thus proposes an evaluation method based on an exhaustive exploration of the performance lattice. The benefit of parameterized definitions is that the same method applies in different contexts each with its own performance envelope.

3. The Benchmark Programs

For our evaluation of Typed Racket, we curated a suite of twelve programs. They are representative of actual user code yet small enough so that exhaustive exploration of the performance lattice remains tractable. The benchmarks are either based on third-party libraries or scripts sourced from the original developer or the Racket package repository.

3.1 Overview

The table in figure 1 lists and summarizes our twelve benchmark programs. For each, we give an approximate measure of the program's size, a diagram of its module structure, and a worst-case measure of the contracts created and checked at runtime.

Size is measured by the number of modules and lines of code (LOC) in a program.[3] Crucially, the number of modules also determines the number of gradually-typed configurations to be run when testing the benchmark, as a program with *n* modules can be gradually-typed in 2^n possible configurations. Lines of code is less important for evaluating macro-level gradual typing, but gives a sense of the overall complexity of each benchmark. Moreover, the Type Annotations LOC numbers are an upper bound on the annotations required at any stage of gradual typing because each typed module in our experiment fully annotates its import statements. In practice, only imports from untyped modules require annotations.

[3] We measured lines of code using the sloccount utility.

The column labeled "Other LOC" measures the additional infrastructure required to run each project for all typed-untyped configurations. This count includes project-wide type definitions, typed interfaces to untyped libraries, and any so-called type adaptor modules (see below).

Finally, the module structure graphs show a dot for each module in the program and an arrow from one module to another when the module at the arrow tail imports definitions from the module at the arrow head. When one of these modules is typed and the other untyped, the imported definitions are wrapped with a contract to ensure type soundness. To give a sense of how "expensive" the contracts at each boundary are, we have colored and thickened arrows to match the absolute number of times any contracts formed at each boundary were checked. The color scale is shown below the table.

The colors fail, however, to show the cost of protecting data structures imported from another library or factored through an adaptor module. For example, the `kcfa` graph has many thin black edges because the modules only share data definitions; the expensive contracts are all formed within a module rather than along a boundary. Instead, we include the column "% Missing" to give the proportion of observed contract checks that are not show in the module graphs.

3.1.1 Adaptor Modules

Type adaptor modules are specialized typed interfaces to untyped code that are used when an untyped data definition and the data's typed clients are part of the same configuration. The adaptor, itself a typed module, exports annotated versions of all bindings in the untyped data definition. Typed clients are set up to import exclusively from the type adaptor, bypassing the original data definition. Untyped clients continue to use the untyped file.

Figure 2 illustrates the basic problem that type adaptors solve. The issue is that Racket structure types (record type definitions) are *generative*; each assignment of a type annotation to an untyped structure creates a new "black box" definition. This means that two syntactically-identical type assignments to the same structure are incompatible. Using an adaptor ensures that only one canonical type is generated for each structure, as illustrated in the right half of figure 2.

Strictly speaking, type adaptor modules are not necessary. It is possible to modify the design of imports for any given configuration so that a single typed module declares and re-exports type annotations for untyped data. This alternative presents a non-trivial challenge, however, when trying to synthesize the 2^n gradually-typed configurations from a fully-untyped and fully-typed version of each benchmark.

The layer of indirection provided by adaptors solves this issue and reduces the number of type annotations needed at boundaries because all typed clients can reference a single point of control.[4] Therefore we expect type adaptor modules to be of independent use to practitioners.

3.2 Program Descriptions

This section briefly describes each benchmark, noting the dependencies and required adaptor modules. Unless otherwise noted, the benchmarks rely only on core Racket libraries and do not use adaptor modules.

Sieve This program finds prime numbers using the Sieve of Eratosthenes and is our smallest benchmark. It consists of two modules: a tiny streams library and a script implementing the Sieve using streams.

[4] In our experimental framework, type adaptors are available to all configurations as library files.

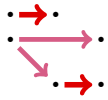
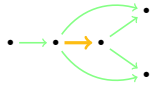
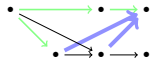
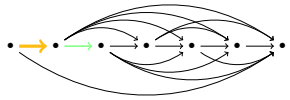
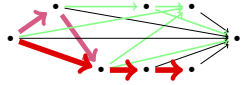
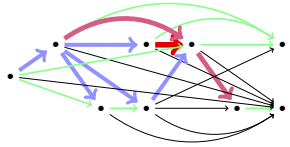
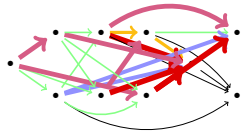
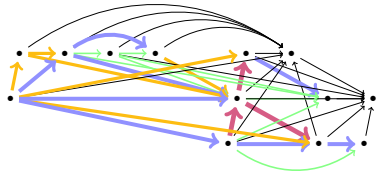
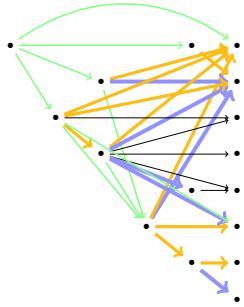
Project name	# Modules	Untyped LOC	Type Ann. LOC	Other LOC	Module structure	% Missing
`sieve`	2	35	17	0		0
`morse-code`	4	216	29	0		0
`mbta`	4	369	77	89		79
`zo-traversal`	5	1404	285	214		99
`suffixtree`	6	545	125	40		97
`lnm`	6	501	120	62		46
`kcfa`	7	248	47	141		99
`snake`	8	161	50	27		93
`tetris`	9	305	71	38		99
`synth`	10	837	142	33		47
`gregor`	13	996	164	103		78
`quad`	16	6722	300	241		45

Figure 1: The software characteristics of the benchmarks

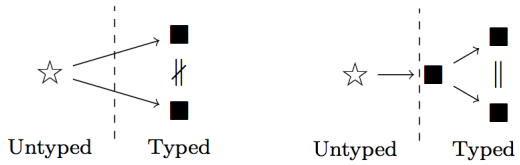


Figure 2: Inserting a type adaptor

Morse code The `morse-code` script is adapted from a morse code training program[5]. The original program plays a morse code audio clip, reads the keyboard for user input, and scores the input based on its Levenshtein distance from the correct answer. Our benchmark tests generating morse code strings and running the Levenshtein algorithm on a list of frequently-used English words.

MBTA The `mbta` program implements a server that asynchronously responds to path queries about a graph representation of Boston's public transit system. The graph representation is implemented using a third-party, untyped library. The latter introduces a typed-untyped boundary even in the "completely typed" case.

ZO Traversal The `zo-traversal` script provides a tool for exploring Racket bytecode structures and counts the frequency of AST nodes. The script operates on the Racket compiler's untyped zo data structures. Since these data structures are not natively supported in Typed Racket, even the completely typed program incurs some dynamic overhead.

Suffixtree The `suffixtree` library implements a longest-common-substring algorithm using Ukkonen's suffix tree algorithm. While the library has minimal external dependencies, it calls for one adaptor module for the algorithm's internal data structures.

L-NM This script analyzes the measurements included in this paper and generates figures 4 and 5. Most of this benchmark's running time is spent generating figures using Typed Racket's `plot` library, so the *untyped* version of this progam is noticably less performant on large datasets. This program relies on an untyped image rendering library and uses two adaptor modules.

K-CFA The `kcfa` program implements a simple control flow analysis for a lambda calculus. The language definitions and analysis are spread across seven modules, four of which require adaptors because they introduce new datatypes.

Snake This program is based on a contract verification benchmark[6] by Nguyên et al. [15]. It implements a game where a growing and moving snake tries to eat apples while avoiding walls and its own tail. Our benchmark, like Nguyên's, runs a pre-recorded history of moves altering the game state and does not display a GUI. We use one adaptor module to represent the game datatypes, but otherwise the program is self-contained.

Tetris This program is taken from the same benchmark suite as `snake` [15] and implements the eponymous game. Like `snake`, the benchmark runs a pre-recorded set of moves. Using it here requires one adaptor module.

Synth The `synth` benchmark[7] is a sound synthesis example from St-Amour et al.'s work on feature-specific profiling [22]. The program consists of nine modules, half of which are from Typed Racket's array library. In order to run these library modules in all typed-untyped configurations we created an adaptor module for the underlying array data structure.

Gregor This benchmark consists of thirteen modules and stress-tests a date and time library. The original library uses a library for ad-hoc polymorphism that is not supported by Typed Racket. We get around this limitation by monomorphizing the code and removing `gregor`'s string parsing component. The benchmark uses two adaptor modules and relies on a small, untyped library for acquiring data on local times.

Quad This project implements a type-setting library. It depends on an external constraint satisfaction solver library (to divide lines of text across multiple columns) and uses two adaptor modules.

4. Evaluating Typed Racket

Measuring the running time for the performance lattices of our chosen dozen benchmarks means compiling, running, and timing hundreds of thousands of configurations. Each configuration is run 30 times to ensure that the timing is not affected by random factors; some configurations take minutes to run. Presenting and analyzing this wealth of data poses a separate challenge all by itself.

This section presents our measurements. The first subsection discusses one benchmark in detail, demonstrating how we create the configurations, how the boundaries affect the performance of various configurations, and how the Typed Racket code base limits the experiment. The second subsection explains how we present the data in terms of the definitions of section 2. The last subsection discuss the results for all benchmarks.

Experimental setup Due to the high resource requirements of evaluating the performance lattices, experiments were run on a cluster consisting of a Machine A with 12 physical Xeon E5-2630 2.30GHz cores and 64GB RAM, Machine B with 4 physical Core i7-4790 3.60GHz cores and 16GB RAM, and a set of Machines C with identical configurations of 20 physical Xeon E5-2680 2.8GHz cores with 64GB RAM. All machines run a variant of Linux. The following benchmarks were run on machine A: `sieve` and `kcfa`. The following were run on machine B: `suffixtree`, `morse-code`, and `lnm`. The following were run on machine C: `snake`, `synth`, `tetris`, `gregor`, and `quad`. For each configuration we report the average of 30 runs. All of our runs use a single core for each configuration with green threads where applicable. We did some sanity checks and were able to validate that performance differentials reported in the paper were not affected by the choice of machine.

4.1 Suffixtree in Depth

To illustrate the key points of the experiments, this section describes one of the benchmarks, `suffixtree`, and explains the setup and its timing results in detail.

`Suffixtree` consists of six modules: `data` to define label and tree nodes, `label` with functions on suffixtree node labels, `lcs` to compute Longest-Common-Subsequences, `main` to apply lcs to data, `structs` to create and traverse suffix tree nodes, `ukkonen` to build suffix trees via Ukkonen's algorithm. Each module is available with and without type annotations. Each configuration thus links six modules, some of them typed and others untyped.

Typed modules require type annotations on their datatypes and functions. Modules provide their exports with types, so that the type checker can cross-check modules. A typed module may import values from an untyped module, which forces the corresponding `require` specifications to come with types. Consider this example:

```
(require
  (only-in "label.rkt"
```

[5] `https://github.com/jbclements/morse-code-trainer`

[6] `https://github.com/philnguyen/soft-contract`

[7] `https://github.com/stamourv/synth`

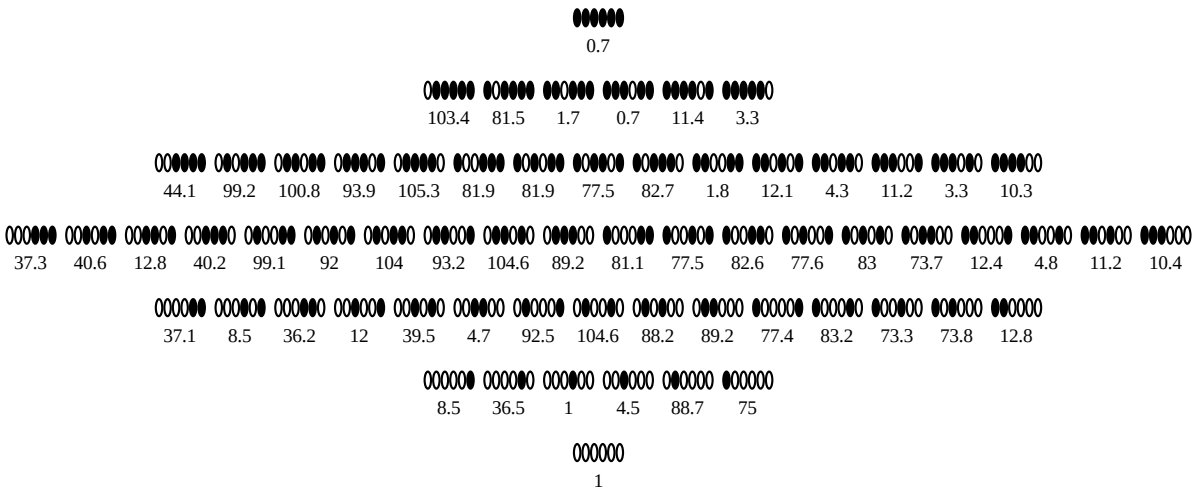


Figure 3: Performance lattice (labels are slowdowns).

```
            make-label
            sublabel
            ...
            vector->label/s))
```

The server module is called `label.rkt`, and the client imports specific values, e.g., `make-label`. This specification is replaced with a `require/typed` specification where each imported identifier is typed:

```
(require/typed/check "label.rkt"
 [make-label
  (-> (U String (Vectorof (U Char Symbol))) Label)]
 [sublabel
  (case->
    (-> Label Index Label)
    (-> Label Index Index Label))]
 ...
 [vector->label/s
  (-> (Vectorof (U Char Symbol)) Label)])
```

The types in a `require/typed` specification are compiled into contracts for the values that flow into the module. For example, if some imported variable is declared to be a `Char`, the check `char?` is performed as the value flows across the module boundary. Higher-order types (functions, objects, or classes) become contracts that are wrapped around the imported value and which check future interactions of this value with its context.

The performance costs of gradual typing thus consist of allocation of wrapper and run-time checks. Moreover, the compiler has to assume that any value could be wrapped, so cannot generate direct field access code as would be done in a statically typed language.

Since our evaluation setup calls for linking typed modules to both typed and untyped server modules, depending on the configuration, we replace `require/typed` specifications with `require/typed/check` versions. This new syntax can determine whether the server module is typed or untyped. It installs contracts if the server module is untyped, and it ignores the annotation if the server module is typed. As a result, typed modules function independently of the rest of the modules in a configuration.

Performance Lattice. Figure 3 shows the performance lattice annotated with the timing measurements. The lattice displays each of the modules in the program with a shape. A filled black shape means the module is typed, an open shape means the module is untyped. The shapes are ordered from left to right and correspond to the modules of `suffixtree` in alphabetical order: `data`, `label`, `lcs`, `main`, `structs`, and `ukkonen`.

For each configuration in the lattice, the ratio is computed by dividing the average timing of the typed program by the untyped average. The figure omits standard deviations as they small enough to not affect the discusion.

The fully typed configuration (top) is *faster* than the fully untyped (bottom) configuration by around 30%, which puts the typed/untyped ratio at 0.7. This is easily explained by Typed Racket's optimizations such as specialization of arithmetic operations, improved field accesses, and elimination of some bounds checks [26]. When the optimizer is turned off, the ratio goes back up to 1.

Sadly, the performance improvement of the typed configuration is the only good part of this benchmark. Almost all partially typed configurations exhibit slowdowns between 0.7x and 105x. Inspection of the lattice suggests several points about these slowdowns:

- Adding type annotations to the `main` module neither subtracts or adds much overhead because it is a driver module that is not coupled to other modules.
- Adding types to any of the workhorse modules—`data`, `label`, or `structs`—while leaving all other modules untyped causes slowdown of at least 35x. These modules make up tightly coupled clique. Laying down a type-untyped boundary to separate this clique causes many crossings of values, with associated contract-checking cost.
- Inspecting `data` and `label` further reveals that the latter depends on the former through an adaptor module. This adaptor introduces a contract boundary when either of the two modules is untyped. When both modules are typed but all others remain untyped, the slowdown is reduced to about 13x.

  The `structs` module depends on `data` in the same fashion. Because `structs` also depends on `label`, the configuration

in which both `structs` and `data` are typed still has a large slowdown. When all three modules are typed, the slowdown is reduced to about 5x.

- Finally, the configurations close to the worst slowdown case are those in which the `data` module is left untyped but several of the other modules are typed. This makes sense given the coupling noted above; the contract boundaries induced between the untyped `data` and other typed modules slow down the program. The module structure diagram for `suffixtree` in figure 1 corroborates the presence of this coupling. The rightmost node in that diagram corresponds to the `data` module, which has the most in-edges in that particular graph. We observe a similar kind of coupling in the simpler `sieve` example, which consists of just a data module and its client.

The performance lattice for `suffixtree` is bad news for gradual typing. It exhibits performance "valleys" in which a maintenance programmer can get stuck. Consider starting with the untyped program, and for some reason choosing to add types to `label`. The program slows down by 88x. Without any guidance, a developer may choose to then type `structs` and see the program slow down to 104x. After that, typing `main` (104x), `ukkonen` (99x), and `lcs` (103x) do little to improve performance. It is only when all the modules are typed that performance improves (0.7x).

4.2 Experimental Results

Figures 4 and 5 summarize the results from exhaustively exploring the performance lattice of our twelve benchmarks. Each row reports the results for one program. On the left, a table spells out the typed/untyped ratio, the maximum overhead, the average overhead, and the number of 300-deliverable and 300/1000-usable configurations. This is followed by three graphs that show how many configurations impose an overhead between 1x and 20x for three values of L: 0, 1, and 2.

The typed/untyped ratio is the slowdown or speedup of fully typed code over untyped code. Values smaller than 1 indicate a speedup due to some of the Typed Racket optimizations. Values larger than 1 are slowdowns caused by interaction with untyped parts of the underlying Racket runtime. The ratios ranges between 0.28x (`lnm`) and 3.22x (`zo-traversal`).

The maximum overhead is computed by finding the running time of the slowest configuration and dividing it by the running time of the untyped version. The average overhead is obtained by computing the average over all configurations (excluding the top and bottom ones) and dividing it by the running time of the untyped configuration. Maximum overheads range from 1.25x (`lmn`) to 168x (`tetris`). Average overheads range from 0.6x (`lmn`) to 68x (`tetris`). The slowdowns reported in the partially typed configurations come from contracts and checks performed as untyped code interacts with typed code.

The 300-deliverable and 300/1000-usable counts are computed for L=0. Next to each count, we report (in parentheses) the count divided by the number of configurations.

The three cumulative performance graphs are read as follows. The x-axis represents the slowdown over the untyped program (from 1x to 20x). The y-axis is a count of the number of configurations (from 0 to 2^n) scaled so all graphs are the same height. The blue curves show how many configurations have less than a given slowdown. They are, by definition, monotonically increasing because the more overhead is allowed, the more configurations satisfy the condition. The "ideal" result would be a flat line at a graph's top. Such a result would mean that all configuration are as fast (or faster) as the untyped one. The "worst" scenario is a flat line at the graph's bottom, indicating that all configuration are more than 20x slower than the untyped one. For ease of comparison between graphs, a dashed (red) horizontal line indicates 60% point of all configurations. As a rule of thumb, curves that climb faster are better than curves with a smaller slope.

Our method defines N-deliverable and N/M-usable as key functions for measuring the quality of a gradual type system. The cumulative performance graphs display the values of parameters N and M as, respectively, a green and a yellow vertical line. For this experiment we have chosen values of 300% and 1000%. These values are rather liberal and we expect that most production contexts would not tolerate anything higher than 200% (and some would object to any slowdown). Thus, for any program, the value of 300-deliverable is the value of the y-axis where the blue line intersects the green one. This is how many configurations are deliverable. Similarly for 300/1000-usable, its value is the difference of the intercepts. Higher values for both of these measures are better.

Lastly, the figures show cumulative performance graphs for different values of L (from 0 to 2). The interpretation of L = 1 and 2 are as follows. For each configuration, we search the entire space of $O(n!\text{-}(n\text{-}L)!)$ reachable configurations to find a neighbor with usable running time. If the top configuration (fully typed) is reachable, it is included in the set. Clearly the number of steps is bounded by the height of the performance lattice. Thus, for example, `sieve` which has only two modules can get an ideal result in one step.

4.3 Interpretation

As mentioned, the ideal shape for these curves is a flat line at the top of the y-axis. Of course the dynamic checks inserted by gradual type systems make this ideal difficult to achieve even with type-driven optimizations, so the next-best shape is a steep vertical line reaching the 100% count at a low x-value. A steep slope from the 1x point means that a large proportion of all configurations run within a small constant overhead. For lines with lower gradients this small constant must be replaced with a larger overhead factor for the same proportion of configurations to qualify as acceptable.

Given the wide x-axis range of overhead factors, we would expect that only the leftmost quarter of each graph shows any interesting vertical slope. Put another way, if it were true that Typed Racket's sound gradual typing is practical but requires tuning and optimization, the data right of the 10x point should be nearly horizontal and well above the red dashed line for L=0.[8] This shape would suggest that although a small percentage of configurations suffer an order of magnitude slowdown, the performance of most gradually-typed configurations is within a (large) constant factor.

We now describe the shape of the results for each benchmark. Our procedure is to focus on the left column, where L=0, and to consider the center and right column as rather drastic countermeasures to recover performance.

Sieve At L=0, the `sieve` benchmark is dead in the water, as both partially typed configuration have more than 20x overhead. Increasing L, however, shows unsurprisingly that the fully typed configuration is one step away.

Morse code The `morse-code` benchmark shows acceptable performance. At L=0 three configurations perform at least as well as the untyped program, and the maximum overhead is below 2x. Increasing L raises the y-intercept of the lines, which matches the observation given that the fully-typed `morse-code` runs faster than the original program.

MBTA The `mbta` benchmark is nearly a steep vertical line, except for one flat area. This implies that a boundary (or group of boundaries) accounts for a 3x slowdown, such that the set of configurations where these boundaries connect typed and untyped mod-

[8] Increasing L should remove pathologically-bad cases.

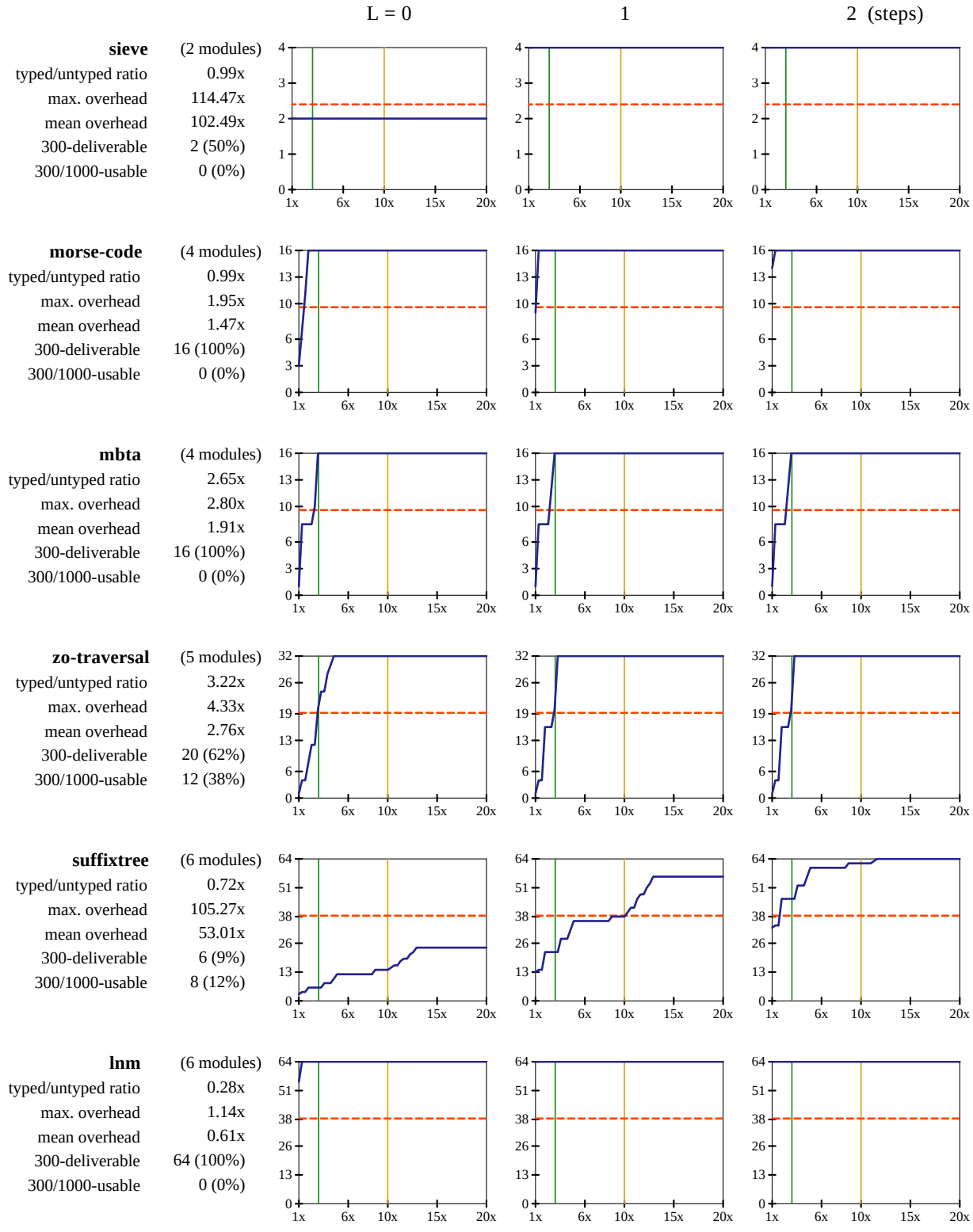


Figure 4: *L*-step *N/M*-usable results for the first 6 benchmarks. The x-axes measure overhead and the y-axes count configurations.

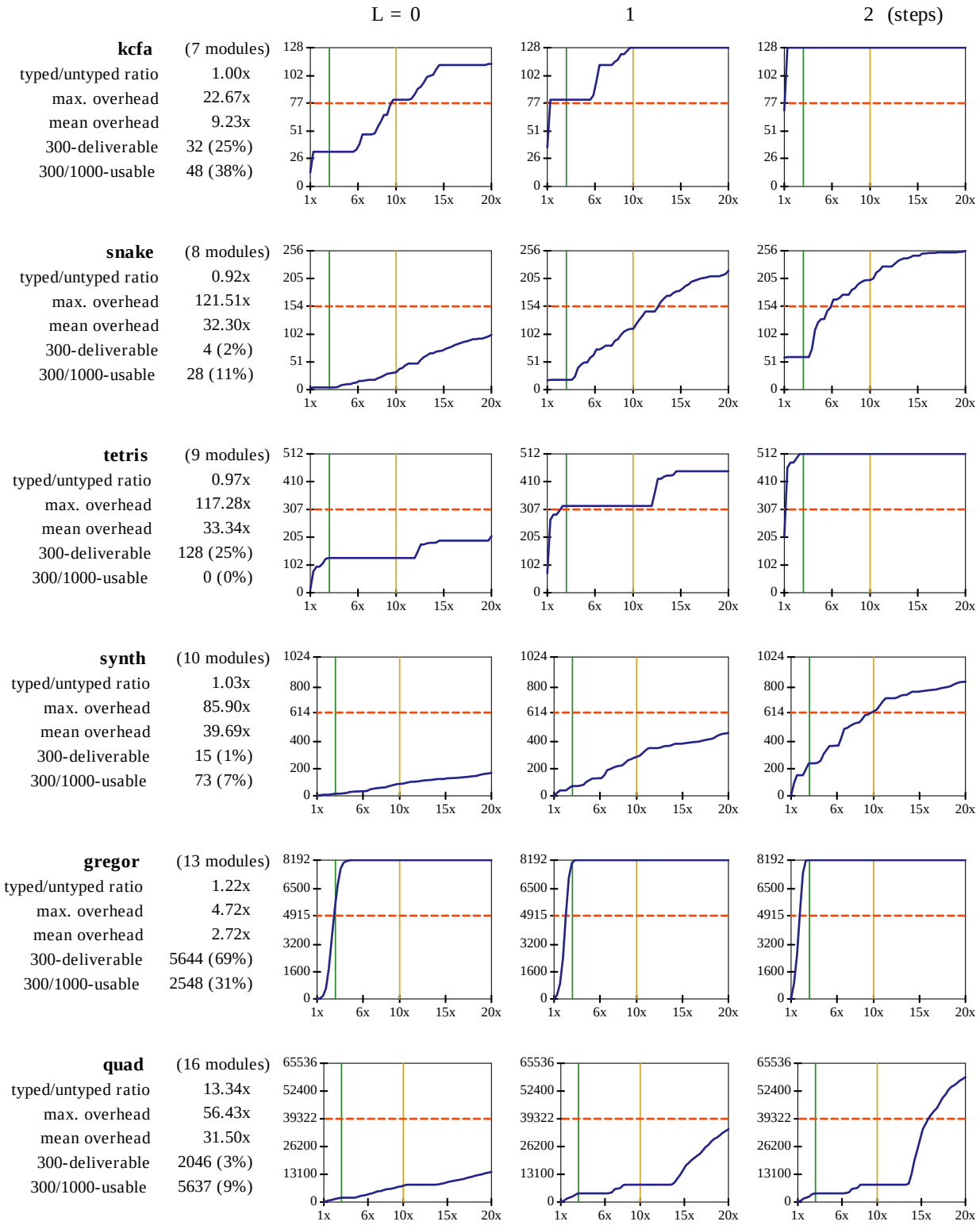


Figure 5: *L*-step *N*/*M*-usable results for the remaining benchmarks

ules all experience similar overhead. Indeed, we found that applying a typed wrapper to the untyped graph library that `mbta` depends on caused a 300% increase in the total number of contract checks.[9] This also explains why the fully typed configuration is not 2-deliverable.

ZO Traversal The curves for `zo-traversal` are fairly steep, but not as drastic as `morse-code` or `mbta`. Half the configurations suffer a 2x overhead, even when L increases. This behavior is again explained by one expensive boundary: when the underlying bytecode representation is wrapped in a contract, the program incurs an additional 500,000 contract checks. Hence projects that are "untyped enough" to avoid creating this boundary perform significantly better.

Suffixtree At L=0, `suffixtree` is the worst of our benchmarks. Over half the gradually-typed configurations are not usable at any realistic overhead. Increasing L, however, improves this picture. Thus although most configurations suffer large performance overhead, they are, in theory, close to a configuration with better performance. Nevertheless there are still too many bad configurations for comfort.

LNM The shape of the `lnm` graphs is ideal. The sharp vertical line at L=0 indicates that gradual typing introduces only a small overhead compared to the untyped program. Indeed, the summary for `lnm` confirm that the maximum overhead is 1.14x slower than the untyped baseline. Furthermore, the fully typed performance is very good. Likely, this is due to the heavy use of Racket's plotting library, which is typed. This also suggest that the untyped program suffers from a performance penalty due to contracts.

K-CFA The `kcfa` benchmark has a jagged shape, implying that N/M-usability is not a helpful tradeoff for this program. At L=0, selecting an N strongly influences the proportion of acceptable configurations for small values of M. This is especially true for N between 1x and 6x, and remains true even after increasing L to 1; however at L=2 it is possible to find N-deliverable configurations from any configuration.

Snake The `snake` benchmark performs well when fully typed, but most partially typed configurations suffer from more than 20x overhead. Increasing L helps somewhat, but still one must accept 6x overhead before the majority of configurations qualify as usable.

Tetris Like `suffixtree`, the `tetris` benchmark is a success story for increasing L. When L=0 we see that most of the modules are more than 20x worse than the untyped program. The good configurations are apparently spread throughout the lattice so that they are easily reachable in two steps of typing.

Synth The `synth` benchmark is similar to `snake`. Over half the configurations suffer an overhead of more than 20x. Increasing L does increase the slopes of the lines, meaning a larger number of configurations become usable for a fixed N/M pair, but gradual typing still introduces a large overhead. Even at L=2 only 30% of all configurations lie in reach of a point with at most 3x slowdown.

Gregor Despite being a large benchmark, `gregor` performs reasonably well even when gradually-typed. The worst-case slowdown of 5.2x is quite good compared to the other large benchmarks, and the steep vertical slope is encouraging.

Quad Only a small proportion of `quad` variations are 300-deliverable or 300/1000-usable for any value of `L`. The `L=2` graph does show a large spike in the number of 1500-usable configurations, but this fact is not likely to help developers exploring the lattice.

[9] From ∼100 to ∼300 checks

5. Sound Gradual Typing is Dead

Unsound type systems are useful. They find bugs at compile-time, and an IDE can use them to assist programmers. Sound type systems are meaningful in addition. A soundly typed program cannot go wrong, up to a well-defined set of run-time exceptions [28]. When a typed program raises an exception, the exception message can (usually) pinpoint the location of the problem in the program source. Hence sound types are one of the best forms of documentation and specification around. In the context of a sound type system, the type of a well-named function or method often explains (almost) as much as an inspection of the code.

From this description, it is clear why programmers eventually wish to annotate programs in untyped languages with types and, ideally, with sound types. Types increase a programmer's productivity, and sound types greatly help with testing, debugging, and other bug-related maintenance tasks. Hence sound gradual typing seems to be such a panacea. The problem is, however, that the cost of enforcing soundness appears to be overwhelming according to our measurements.

In general, the graphs in figures 4 and 5 clarify how few partially typed configurations are usable by developers or deliverable to customers. For almost all benchmarks, the lines are below the (red) horizontal line of acceptability. Even with extremely liberal settings for *N* and *M*, few configurations are *N*-deliverable or *N/M*-usable. Worse, investing more effort into type annotation does not seem to pay off. In practice, converting a module takes a good portion of a workday, meaning that setting *L* to 2 is again a liberal choice. But even allowing the conversion of two additional modules *and* the unrealistic assumption that the developer picks two modules best-suited to improve performance does not increase the number of acceptable configurations by much. Put differently, the number of *L*-step *N/M*-acceptable configurations remains small with liberal choices for all three parameters.

Like Marc Antony, we come here to bury sound gradual typing, not to praise it. Sound gradual typing is dead.

5.1 Threats to Validity of Conclusion

Our judgment is harsh and fails to acknowledge potential weaknesses in our evaluation method and in our results.

First, our benchmarks are relatively small. The two largest ones consist of 13 and 16 modules, respectively. Even these benchmarks pose challenges to our computing infrastructure because they require timing 2^{13} and 2^{16} configurations 30 times each. Running experiments with modules that consist of many more modules would be impractical. Our results might be less valid in the context of large programs, though practical experience using Typed Racket suggests otherwise. To make the experiment feasible even at the sizes we use in this paper, we run the larger benchmarks using multiple cores and divide up the configurations amongst the cores. Each configuration is run in a single process running a separate instance of the Racket VM pinned to a single core. However, this parallelism may introduce confounding variables due to, for example, shared caches or main memory. We have attempted to control for this case and, as far as we can tell, executing on an unloaded machine does not make a significant difference to our results.

Second, several of our benchmarks import some modules from Racket's suite of libraries that remain untyped throughout the process, including for the fully typed configuration. While some of these run-time libraries come in the trusted code base—meaning Typed Racket knows their types and the types are not compiled to contracts—others are third-party libraries that impose a cost on all configurations. In principle, these interfaces might substantially contribute to the running-time overhead of partially typed configurations. But, given the low typed/untyped ratios, these libraries are unlikely to affect our conclusions.

Third, our method imagines a particularly *free* approach to annotating programs with types. By "free" we mean that we do not expect software engineers to add types to modules in any particular order. Although this freedom is representative of some kind of maintenance work—add types when bugs are repaired and only then—a team may decide to add types to an entire project in a focused approach. In this case, they may come up with a particular kind of plan that avoids all of these performance traps. Roughly speaking, such a plan would correspond to a specific path from the bottom element of the performance lattice to the top element. Sadly our current measurements suggest that almost all bottom-to-top paths in our performance lattices go through performance bottlenecks. As the `suffixtree` example demonstrates, a path-based approach depends very much on the structure of the module graph. We therefore conjecture that some of the ideas offered in the conclusion section may help such planned, path-based approaches.

Fourth, we state our judgment with respect to the current implementation technology. Typed Racket compiles to Racket, which uses rather conventional compilation technology. It makes no attempt to reduce the overhead of contracts or to exploit contracts for optimizations. It remains to be seen whether contract-aware compilers can reduce the significant overhead that our evaluation shows. Nevertheless, we are convinced that even if the magnitude of the slowdowns are reduced, some pathologies will remain.

6. The State of the Related Work

Gradual typing is a broad area teeming with both theoretical and practical results. This section focuses on implementations rather than formal models, paying special attention to performance evaluation of gradual type systems.

6.1 Sound Gradual Type Systems

Gradual typing has already been applied to a number of languages: Python [27], Smalltalk [2], Thorn [6] and TypeScript [17, 18]. None of the projects report on conclusive studies of gradual typing's impact on performance.

The authors of Reticulated Python recognized the performance issues of gradual typing and designed the language to allow the exploration of efficient cast mechanisms. However, Vitousek et al. note that "Reticulated programs perform far worse than their unchecked Python implementations" and that their `slowSHA` program exhibits a "10x slowdown" compared to Python [27, pg. 54].

Gradualtalk's evaluation is primarily qualitative, but Allende et al. have investigated the overhead of several cast-insertion strategies on Gradualtalk microbenchmarks and on two macrobenchmarks [4]. In addition, Allende et al. [3] investigated the effect of confined gradual typing—an approach in which the programmer can instruct the type system to avoid higher-order wrapping where possible—in Gradualtalk on microbenchmarks. These efforts evaluate the cost of specific features, but do not represent the cost of the whole gradual typing process.

Safe TypeScript comes with an evaluation on the Octane benchmarks ported to TypeScript. Unlike our lattice-based approach, their evaluation compares only the performance of the fully untyped and fully typed programs. For Safe TypeScript, Rastogi et al. report slowdowns in unannotated programs in a "range from a factor of 2.4x (splay) to 72x (crypto), with an average of 22x" [17, pg. 178]. On fully typed programs, the overhead is "on average only 6.5%" [17, pg. 178].

Thorn combines a sound type system with an optional type system, allowing programmers to choose between so-called concrete types and like types [6]. StrongScript follows Thorn's lead by adding a sound type system (with a limited form of higher-order wrappers) to TypeScript. Thorn had a minimal performance evaluation which showed that by sprinkling a few type annotations over toy benchmarks speed ups between 3x and 6x could be obtained [29]. Richards et al. use the same microbenchmark suite as Safe TypeScript and compare the runtimes of type-erased and fully-typed versions using their optimizing compiler. They report "no benchmarks demonstrated slowdown outside of noise" (and up 20% speed ups) on the fully-typed versions [18, pg. 97]. In our lattice terminology, the StrongScript comparison reports typed/untyped ratios only. The performance of intermediate states are not evaluated.

6.2 Optional Type Systems

Optional typing is an old idea whose roots can be traced as far back as MACLISP, which allowed users to declare (unchecked) type specifications [14, §14.2] in an otherwise untyped language. The flavor of these annotations, and those in Lisp descendants such as Common Lisp, differ from the contemporary view of optional types as statically-checked annotations for software maintenance. In Lisp systems, these annotations are used for compiler optimizations and dynamic checking.

Pluggable type systems are a closely related idea, exemplified by Strongtalk [9], and also belong to the unsound camp. Recent implementations, e.g. Papi et al.'s work for Java [16], layer additional typed reasoning on top of existing typed languages rather than untyped languages.

Optional type systems in the contemporary sense exist for Clojure [8], Lua [13], Python[10], PHP[11], ActionScript[12], Dart[13], and JavaScript[14] [5]. Since the type annotations in these systems are unsound for typed-untyped interoperation, they incur no runtime overhead from proxy wrapping or dynamic checks meaning there is no need for a comprehensive evaluation such as the one suggested in this paper.

Previous publications have, however, investigated the performance impact of optional typing with respect to compiler optimizations. Intuitively one would expect that a compiler could use these annotations as hints for generating more preformant code. This intuition is borne out by Chang et al. [10] who reported significant speed up for partially and fully typed ActionScript code over untyped code. But one should take such results with a pinch of salt as they are highly dependent on the quality of the virtual machine used as baseline. Richards et al. [18] report at most 20% speed up for fully typed JavaScript. They ascribe this unimpressive result to the quality of the optimizations implemented in V8. In other words, V8 is able to guess types well enough that providing it with annotations does not help much.

7. Long Live Sound Gradual Typing

In the context of current implementation technology, sound gradual typing is dead. We support this thesis with benchmarking results for *all possible gradual typing scenarios* for a dozen Racket/Typed Racket benchmarks of various sizes and complexities. Even under rather liberal considerations, few of these scenarios end up in deliverable or usable system configurations. Even allowing for additional conversions of untyped portions of the program does not yield much of an improvement.

Our result calls for three orthogonal research efforts. First, Typed Racket is only one implementation of sound gradual typ-

[10] `http://mypy-lang.org/`

[11] `http://hacklang.org/`

[12] `http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7ec7.html`

[13] `https://www.dartlang.org`

[14] `http://flowtype.org`

ing, and it supports only macro-level gradual typing. Before we declare gradual typing completely dead, we must apply our framework to other implementations. The question is whether doing so will yield equally negative results. Safe TypeScript [17] appears to be one natural candidate for such an effort. At the same time, we are also challenged to explore how our evaluation framework can be adapted to the world of micro-level gradual typing, where programmers can equip even the smallest expression with a type annotation and leave the surrounding context untyped. We conjecture that annotating complete functions or classes is an appropriate starting point for such an adaptation experiment.

Second, Typed Racket's implementation may not support runtime checks as well as modern JIT compilers. Typed Racket elaborates into plain Racket, type-checks the result, inserts contracts between typed and untyped modules, and then uses Racket to compile the result [26]. The latter is essentially a conventional ahead-of-time compiler that compiles a function when it is used the first time. One implication is that code from contracts does not get eliminated even if it is re-evaluated for the same value in a plain loop. Clearly, a tracing JIT compiler may eliminate some of the contract overhead in such situations. The Typed Racket team at Indiana University is developing such a tracing compiler backend for Racket [7], dubbed Pycket, and we are looking forward to applying our evaluation framework to this implementation of Typed Racket. Doing so will allow us to validate both the usefulness of the benchmarking framework and the potential of tracing JIT compilers as possible saviors of sound gradual typing.

Third, the acceptance of Typed Racket in the commercial and open-source Racket community suggests that (some) programmers find a way around the performance bottlenecks of sound gradual typing. To expand this community will take the development of both guidelines on how to go about annotating a large system and performance measurement tools that help programmers discover how to identify those components of a gradually-typed configuration that yield the most benefit (per time investment). St-Amour's feature-specific profiler [22] and optimization coaches [23] look promising; we used both kinds of tools to find the reason for some of the most curious performance bottlenecks in our measurements.

In sum, while we accept that the current implementation technology for gradually-typed programming languages falls short of its promises, we also conjecture that a rigorous performance evaluation framework will provide guidance for future research. Above we have spelled out practical directions but even theoretical ideas—such as Henglein's optimal coercion insertion [12] and the collapsing of chains of contracts [21]—may take inspiration from the application of our framework.

Data and Code

We will make our complete benchmark suite, all configurations, and the measurements available jointly with our publication.

Acknowledgments

We thank Matthew Butterick, John Clements, Matthew Might, Phúc C. Nguyễn, Vincent St-Amour, Danny Yoo, and Jon Zeppieri for providing benchmark code bases.

Bibliography

[1] Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *Trans. Programming Languages and Systems* 13(2), pp. 237–268, 1991.

[2] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 2013.

[3] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined Gradual Typing. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 251–270, 2014.

[4] Esteban Allende, Johan Fabry, and Éric Tanter. Cast Insertion Strategies for Gradually-Typed Objects. In *Proc. Dynamic Languages Symposium*, pp. 27–36, 2013.

[5] Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *Proc. European Conf. Object-Oriented Programming*, 2014. http://dx.doi.org/10.1007/978-3-662-44202-9_11

[6] 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 *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 117–136, 2009.

[7] Carl Friedrich Bolz, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Proc. Wksp. on Dynamic Languages and Applications*, 2014.

[8] Ambrose Bonnaire-Sergeant. A Practical Optional Type System for Clojure. Honour's dissertation, University of Western Australia, 2012.

[9] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 215–230, 1993.

[10] Mason Chang, Bernd Mathiske, Edwin Smith, Avik Chaudhuri, Andreas Gal, Michael Bebenita, Christian Wimmer, and Michael Franz. The Impact of Optional Type Information on JIT Compilation of Dynamically Typed Languages. In *Proc. Dynamic Languages Symposium*, pp. 13–24, 2011.

[11] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 48–59, 2002.

[12] Fritz Henglein and Jakob Rehof. Safe Polymorphic Type Inference for a Dynamically Typed Language: Translating Scheme to ML. In *Proc. ACM Intl. Conf. Functional Programming Languages and Computer Architecture*, pp. 192–203, 1995.

[13] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An Optional Type System for Lua. In *Proc. Wksp. on Dynamic Languages and Applications*, pp. 1–10, 2014.

[14] David A. Moon. MACLISP Reference Manual. 1974.

[15] Phúc C. Nguyên, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 139–152, 2014.

[16] Matthew M. Papi, Mahmood Ali, Telmo Louis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical Pluggable Types for Java. In *Proc. Intl. Sym. Software Testing and Analysis*, 2008.

[17] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 167–180, 2015.

[18] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *Proc. European Conf. Object-Oriented Programming*, 2015.

[19] Michael Scriven. The Methodology of Evaluation. Perspectives of Curriculum Evaluation. Rand McNally, 1967.

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

[21] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 365–376, 2010.

[22] Vincent St-Amour, Leif Andersen, and Matthias Felleisen. Feature-specific Profiling. In *Proc. Intl. Conf. on Compiler Construction*, pp. 49–68, 2015.

[23] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 163–178, 2012.

[24] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. In *Proc. European Conf. Object-Oriented Programming*, 2015.

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

[26] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 132–141, 2011.

[27] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. Dynamic Languages Symposium*, pp. 45–56, 2014.

[28] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, pp. 38–94, 1994.

[29] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 377–388, 2010.