Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen

Northeastern University, Boston, USA

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 incremental 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 research on sound gradual typing remains theoretical, the few emerging implementations suffer from performance overheads due to these checks. None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different versions is reported in a synthetic metric that associates runtime overhead to conversion effort. The paper reports on the results of applying the method to Typed Racket, a mature implementation of sound gradual typing, using a suite of real-world programs of various sizes and complexities. Based on these results the paper concludes that, given the current state of implementation technologies, sound gradual typing faces significant challenges. 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.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Performance measures

Keywords Gradual typing, performance evaluation

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 [21, 26] 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 typing, gradual typing provide programmers with soundness guarantees.

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 [12] between typed and untyped modules to enforce soundness. In contrast, *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. The implementation must 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 up to two orders of magnitude [18, 25, 28].

This paper presents a single result: a method for systematically evaluating the performance of a gradual type system. It is illustrated with an application to Typed Racket, a mature implementation of macro-level gradual typing. We find that Typed Racket's cost of soundness is *not* tolerable. If applying our method to other gradual type system implementations yields similar results, then sound gradual typing is dead.

The insight behind the method 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 limited developer resources or access to source code, it may be possible to add types to only a part of the system. Our method must therefore simulate all possibilities. Thus, applying our method to Typed Racket requires annotating all n modules with types. The resulting collection of $2 \cdot n$ modules is then used to create 2^n configurations. The collection of these configurations forms a complete lattice with the

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-3549-2/16/01...
http://dx.doi.org/10.1145/2837614.2837630

untyped configuration at the bottom and the fully typed one at the top. The points in between represent 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 system for gradual typing must time these configurations of a benchmark and extract information from these timings. Section 2 introduces the evaluation method in detail, including the information we retrieve from the lattices and how we parameterize these retrievals. 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. With nine years of development, Typed Racket is the oldest and probably most sophisticated implementation of gradual typing. Furthermore, Typed Racket has also acquired a fair number of users, which suggests adequate performance for these commercial and open source communities. The chosen benchmark programs originate from these communities and range from 150 to 7,500 lines of code. Section 3 presents these benchmarks in detail.

Section 4 presents the results from running all configurations of the Typed Racket benchmarks according to the metrics spelled out in section 2. We interpret the ramifications of these rather negative results in section 5 and discuss the threats to validity of these conclusions. The section also includes our report on a preliminary investigation into the possible causes of the slowdowns in our benchmark configurations.

2. Benchmarking Software Evolution

Our evaluation method is inspired by our previous work on extending functional Typed Racket to the object-oriented aspects of Racket, in which we use a lattice-style approach for a preliminary performance evaluation [25]. By inspecting the entire lattices of typed/untyped configurations of two small game systems, we identified and then eliminated a major performance bottleneck from the implementation. Our previous performance evaluation was conducted in tandem with the design and implementation of Typed Racket, and thus the final results were relatively positive. In contrast, we conduct our current evaluation completely *independently* of Typed Racket's implementation efforts.[1]

Let us re-articulate the salient points from our previous work:

- 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.
- 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 program 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 program 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 the type system does not support. 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 in which 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 boundaries 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 run-time costs may not be tolerable, however, as our previous work observes. In that case, the question is how much more effort 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 an Nx slowdown compared to the completely untyped configuration.

We parameterize this definition over the slowdown factor that a team may consider acceptable. One team may think of a 1.1x slowdown as barely acceptable, while another one may tolerate a slowdown of an order of magnitude [25].

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 an Nx slowdown and no worse than an Mx slowdown compared to the completely untyped configuration.

[1] In terminology borrowed from the education community [20], we conducted a *formative evaluation* while this paper conducts a *summative evaluation* to assess the post-intervention state of the system.

Using the first parameter, we exclude deliverable configurations from the count. The second parameter specifies the positive boundary, i.e., the acceptable slowdown factor for a usable configuration.

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.

This paper thus proposes an evaluation method based on a systematic exploration of the performance lattice. The benefit of parameterized metrics is that every reader can interpret the raw data with his or her own choices for L, N, and M.

3. The Benchmark Programs

For our evaluation of Typed Racket, we use a suite of twelve programs. They are representative of actual user code yet small enough so that an exhaustive exploration of the performance lattice remains tractable.

3.1 Overview

The table in figure 2 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. 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.

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).

The module structure graphs show a dot for each module in the program. An arrow is drawn from module A to module B when module A imports definitions from module B. 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 color arrows to match the absolute number of times contracts at a given boundary are checked. These numbers are independent from the actual configurations.

The colors fail to show the cost of checking 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 column labeled "Adaptors + Libraries" reports the proportion of observed contract checks due to adaptor modules and libraries.

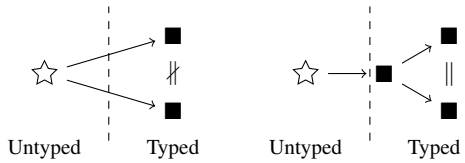


Figure 1: Inserting a type adaptor

3.2 Adaptor Modules

A quirk in Racket's structure-type definitions calls for one twist to an otherwise straightforward setup of benchmark configurations. Consider the following structure-type definition from `gregor`, one of the benchmark programs:

```
(struct DateTime [date time jd])
```

Its evaluation introduces a new class of data structures via a constructor (`DateTime`), a predicate (`DateTime?`), and a number of selectors. A second evaluation creates a disjoint class of structures, meaning the selectors for the first class do not work on the second and vice versa.

If a structure-type definition is exported, a configuration may place the definition in an untyped module and its clients into the typed portion of the program. As explained below, importing a `struct` demands that each client assigns a type to the structure-type definition. Now, when these typed clients wish to exchange instances of these structure types, the type checker must prove that the static types match. But due to the above quirk, the type system assigns generative static types to imported structure types. Thus, even if the developers who annotate the two clients with types choose the same names for the imported structure types, the two clients actually have mutually incompatible static types.

Figure 1 illuminates the problems with the left-hand diagram. An export of a structure-type definition from the untyped module (star-shaped) to the two typed clients (black squares) ensures that the type checker cannot equate the two assigned static types. The right-hand side of the figure explains the solution. We manually add a *type adaptor module*. Such adaptor modules are specialized typed interfaces to untyped code. The typed clients import structure-type definitions and the associated static types exclusively from the type adaptor, ensuring that only one canonical type is generated for each structure type. Untyped clients remain untouched and continue to use the original untyped file.

Adaptor modules also reduce the number of type annotations needed at boundaries because all typed clients can reference a single point of control.[2] Therefore we expect type adaptor modules to be of independent use to practitioners, rather than just a synthetic byproduct of our setup.

3.3 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. We credit program authors in parentheses; except for `sieve`, all programs are independently useful.

Sieve (Ben Greenman) This program finds prime numbers using the Sieve of Eratosthenes and is our smallest benchmark. It contains

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

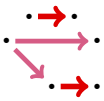
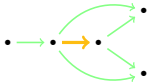
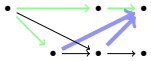
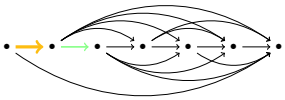
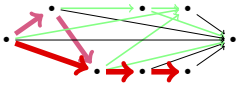
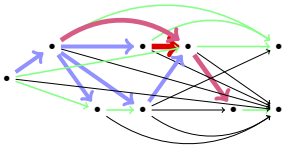
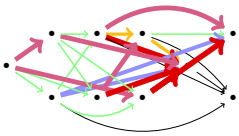
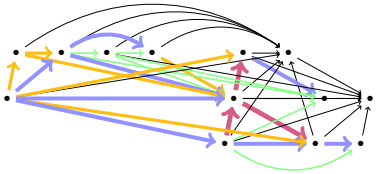
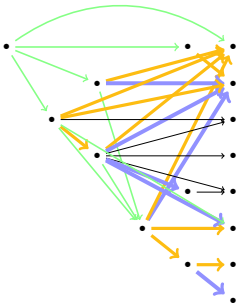
Project name	Modules	Untyped LOC	Type Ann. LOC	Other LOC	Module structure	Adaptors + Libraries
`sieve`	2	35	17	0		0%
`morse-code`	4	216	29	0		0%
`mbta`	4	369	77	89		79%
`zordoz`	5	1404	285	214		99%
`suffixtree`	6	545	125	40		97%
`lnm`	6	501	120	62		46%
`kcfa`	7	248	47	141		99%
`snake`	8	161	50	27		93%
`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 2: Characteristics of the benchmarks

two modules: a streams library and the sieve code. We wrote this benchmark to illustrate the pitfalls of sound gradual typing.

Morse code (John Clements & Neil Van Dyke) This script is adapted from a morse code training program.[3] The original program plays a morse code audio clip, reads keyboard input, and scores the input based on its Levenshtein distance from the correct answer. Our benchmark setup generates morse code strings and runs the Levenshtein algorithm on a list of frequently used words.

MBTA (Matthias Felleisen) The `mbta` program builds a representation of Boston's public transit system and answers reachability queries. It relies on an untyped graph library. The original program responded asynchronously to queries with a server thread. We instead measure a synchronous version of the program to ensure compatibility with Racket's stack-based profiling tools.

Zordoz (Ben Greenman) This tool is used for exploring and counting the frequency of Racket bytecode structures. It 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 (Danny Yoo) This 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.

LNM (Ben Greenman) 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 program is noticeably less performant. This program relies on an untyped image rendering library and uses two adaptor modules.

KCFA (Matt Might) 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 (David Van Horn) This program is based on a contract verification benchmark[4] by Nguyễn et al. [16]. It implements a game where a growing and moving snake tries to eat apples while avoiding walls and its own tail. Our benchmark 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 (David Van Horn) This program is taken from the same benchmark suite as `snake` [16] and implements the eponymous game. Like `snake`, the benchmark runs a pre-recorded set of moves. Using it here requires one adaptor module.

Synth (Vincent St-Amour & Neil Toronto) The `synth` benchmark[5] is a sound synthesis example from St-Amour et al.'s work on feature-specific profiling [23]. 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 create an adaptor module for the underlying array data structure.

Gregor (Jon Zeppieri) 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. Our adaptation instead uses a mono-typed variant of this code and removes the string parsing component. The benchmark uses two adaptor modules and relies on a small, untyped library for acquiring data on local times.

[3] `http://github.com/jbclements/morse-code-trainer`

[4] `http://github.com/philnguyen/soft-contract`

[5] `http://github.com/stamourv/synth`

Quad (Matthew Butterick) This project implements a typesetting library. It depends on an external constraint satisfaction solver library (to divide lines of text across multiple columns) and uses two adaptor modules. The original author provided both untyped and fully typed variants.

4. Evaluating Typed Racket

Measuring the running time for the performance lattices of our benchmarks means compiling, running, and timing 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.

Here we present our measurements in terms of the metrics of section 2. 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 our findings. The last subsection interprets them.

Experimental setup Due to the high resource requirements of evaluating the performance lattices, experiments were run on multiple machines. 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, Machine C with with 4 physical Core i7-3770K 3.50GHz cores and 32GB RAM, and a set of Machines D with identical configurations of 20 physical Xeon E5-2680 2.8GHz cores with 64GB RAM. All machines run a variant of Linux and all benchmarks were run on Racket v6.2. The following benchmarks were run on machine A: `sieve`, `kcfa`, and `gregor`. On machine B: `suffixtree`, `morse-code`, `mbta`, and `lnm`. On machine C: `zordoz` and `quad`. On machine D: `snake`, `synth`, and `tetris`. For each configuration we report the average of 30 runs. All of our runs use a single core for each configuration. We performed sanity checks to validate that performance differentials reported in the paper were not affected by the choice of machine.[6]

4.1 Suffixtree in Depth

To illustrate the key points of the evaluation, 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 substrings, `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 data definitions 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" make-label ...))
```

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 "label.rkt"
 [make-label
  (-> (U String (Vectorof (U Char Symbol))) Label)]
 ...)
```

[6] The scripts that we use to run the experiments are available in our artifact: `http://www.ccs.neu.edu/racket/pubs/#popl15-tfgnvf`

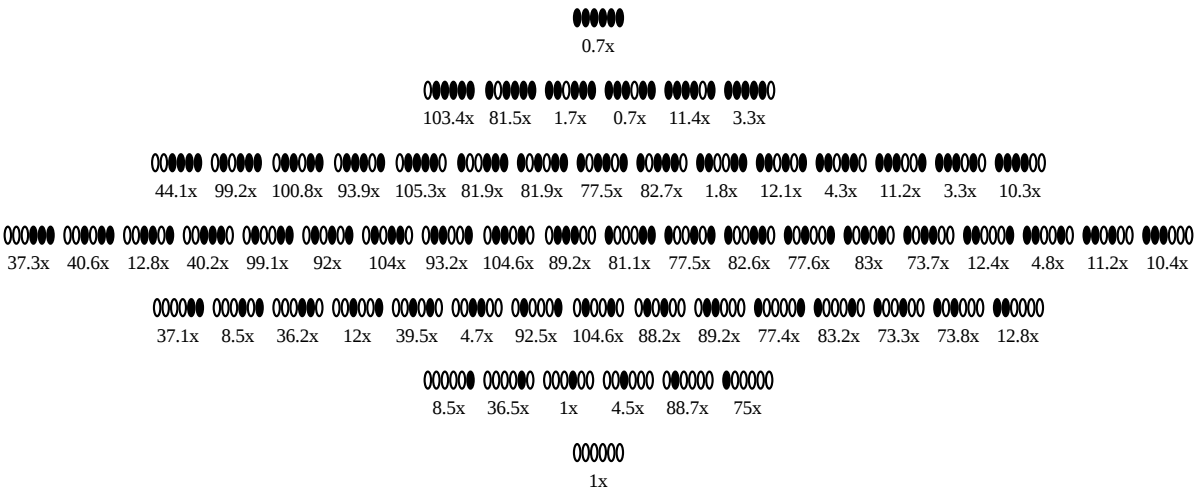


Figure 3: Performance lattice (labels are speedup/slowdown factors)

The types in a `require/typed` form are compiled into contracts for the imported values. 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 wrap the imported value and which check future interactions of this value with its context.

The performance costs of gradual typing thus consist of wrapper allocation and run-time checks. Moreover, the compiler must assume that any value could be wrapped, so it 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 are small enough to not affect the discussion.

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 can be explained by Typed Racket's optimizer, which performs specialization of arithmetic operations and field accesses, and can eliminate some bounds checks [27]. 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 of up to 105x. Inspection of the lattice suggests several points about these slowdowns:

- Adding type annotations to the `main` module neither subtracts nor adds overhead because it is a driver module.
- Adding types to any of the workhorse modules—`data`, `label`, or `structs`—while leaving all other modules untyped causes slowdown of at least 35x. This group of modules are tightly coupled. Laying down a type-untyped boundary to separate elements of this group 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 and additionally on `label`. Thus, 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 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 2 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 a factor of 88x. Without any guidance, a developer may choose to then add types to `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 becomes acceptable again (0.7x).

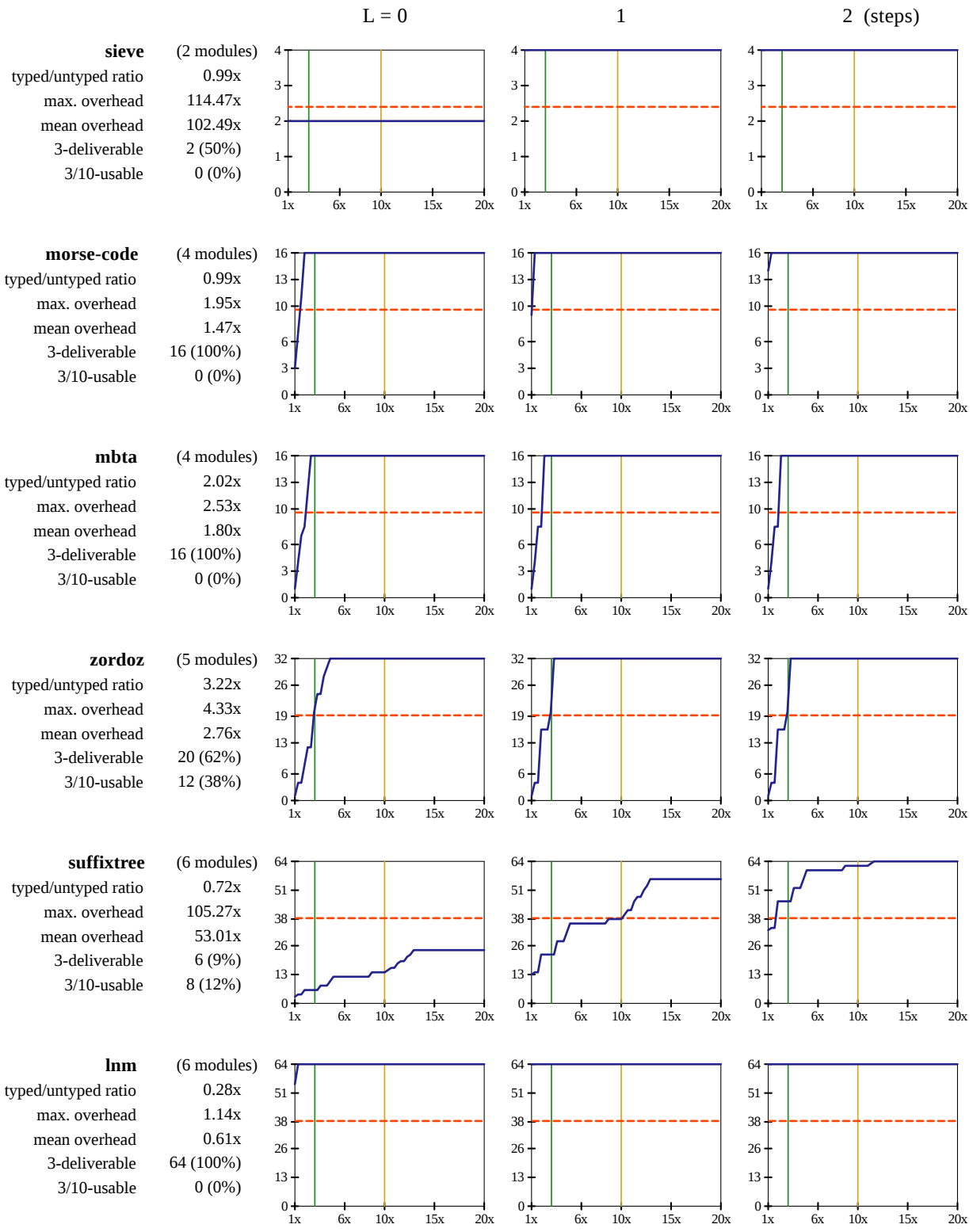


Figure 4: *L*-step *N*/*M*-usable results for the first six benchmarks

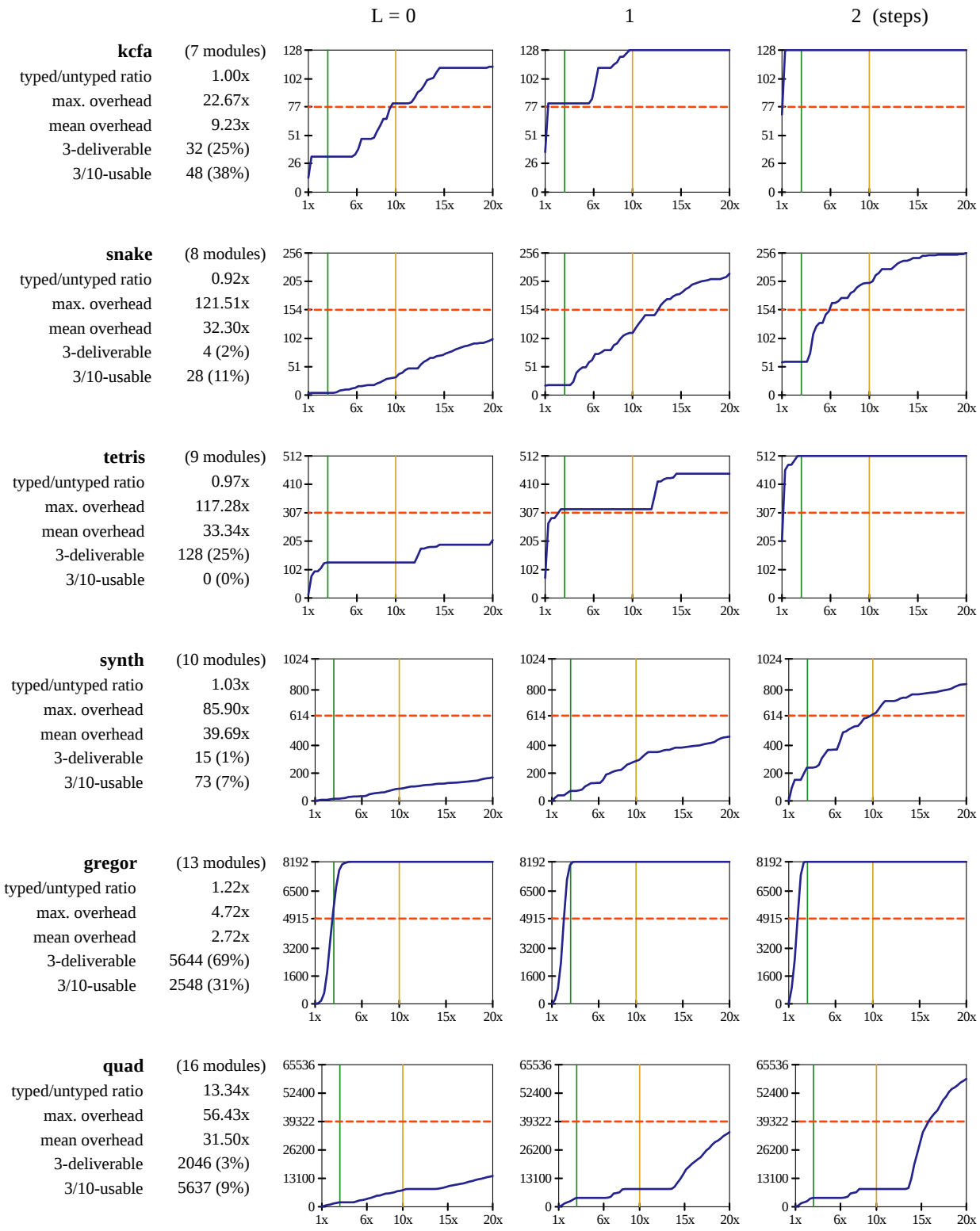


Figure 5: *L*-step *N*/*M*-usable results for the remaining benchmarks

4.2 Reading the Figures

Our method defines the number of L-step N/M-usable configurations as the key metric for measuring the quality of a gradual type system. For this experiment we have chosen values of 3x and 10x for N and M, respectively, and allow up to 2 additional type conversion steps. These values are rather liberal,[7] but serve to ground our discussion.

The twelve rows of graphs in Figures 4 and 5 summarize the results from exhaustively exploring the performance lattices of our benchmarks. Each row contains a table of summary statistics and one graph for each value of L between 0 and 2.

The typed/untyped ratio is the slowdown or speedup of fully typed code over untyped code. Values smaller than 1.0 indicate a speedup due to Typed Racket optimizations. Values larger than 1.0 are slowdowns caused by interaction with untyped libraries or untyped parts of the underlying Racket runtime. The ratios range between 0.28x (`lnm`) and 3.22x (`zordoz`).

The maximum overhead is computed by finding the running time of the slowest configuration and dividing it by the running time of the untyped configuration. The average overhead is obtained by computing the average over all configurations (excluding the fully-typed and untyped configurations) and dividing it by the running time of the untyped configuration. Maximum overheads range from 1.25x (`lnm`) to 168x (`tetris`). Average overheads range from 0.6x (`lnm`) to 68x (`tetris`).

The 3-deliverable and 3/10-usable counts are computed for L=0. In parentheses, we express these counts as a percentage of all configurations for the program.

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 that all graphs are the same height. If L is zero, the blue line represents the total number of configurations with performance no worse than the overhead on the x-axis. For arbitrary L, the blue line gives the number of configurations that can reach a configuration with performance no worse than the overhead on the x-axis in at most L conversion steps.

The ideal result would be a flat line at a graph's top. Such a result would mean that all configurations are as fast as (or faster than) the untyped one. The worst scenario is a flat line at the graph's bottom, indicating that all configurations are more than 20x slower than the untyped one. For ease of comparison between graphs, a dashed (red) horizontal line indicates the 60% point along each project's y-axis.

4.3 Interpretation

The ideal shape is difficult to achieve because of the overwhelming cost of the dynamic checks inserted at the boundaries between typed and untyped code. The next-best shape is a nearly-vertical line that reaches the top at a low x-value. All else being equal, a steep slope anywhere on the graph is desirable because the number of acceptable programs quickly increases at that point.

For each benchmark, we evaluate the actual graphs against these expectations. Our approach 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.[8]

Sieve The flat line at L=0 shows that half of all configurations suffer unacceptable overhead. As there are only 4 configurations in the lattice for `sieve`, increasing L improves performance.

[7] We would expect that most production contexts would not tolerate anything higher than 2x, if that much.

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

Morse code The steep lines show that a few configurations suffer modest overhead (below 2x), otherwise `morse-code` performs well. Increasing L improves the worst cases.

MBTA These lines are also steep, but flatten briefly at 2x. This coincides with the performance of the fully-typed configuration. As one would expect, freedom to type additional modules adds configurations to the 2-deliverable equivalence class.

Zordoz Plots here are similar to `mbta`. There is a gap between the performance of the fully-typed configuration and the performance of the next-fastest lattice point.

Suffixtree The wide horizontal areas are explained by the performance lattice in figure 3: configurations' running times are not evenly distributed but instead vary drastically when certain boundaries exist. Increasing L significantly improves the number of acceptable configuration at 10x and even 3x overhead.

LNM These results are ideal. Note the large y-intercept at L=0. This shows that very few configurations suffer any overhead.

KCFA The most distinctive feature at L=0 is the flat portion between 1x and 6x. This characteristic remains at L=1, and overall performance is very good at L=2.

Snake The slope at L=0 is very low. Allowing L=1 brings a noticeable improvement above the 5x mark, but the difference between L=1 and L=2 is small.

Tetris Each `tetris` plot is essentially a flat line. At L=0 roughly 1/3 of configurations lie below the line. This improves to 2/3 at L=1 and only a few configurations suffer overhead when L=2.

Synth Each slope is very low. Furthermore, some configurations remain unusable even at L=2. These plots have few flat areas, which implies that overheads are spread evenly throughout possible boundaries in the program.

Gregor These steep curves are impressive given that `gregor` has 13 modules. Increasing L brings consistent improvements.

Quad The `quad` plots follow the same pattern as `mbta` and `zordoz`, despite being visually distinct. In all three cases, there is a flat slope for overheads below the typed/untyped ratio and a steep increase just after. The high typed/untyped ratio is explained by small differences in the original author-supplied variants.

5. Quo Vadis Sound Gradual Typing?

Unsound type systems are useful. They document the code, find bugs at compile-time, and enable the IDE to assist programmers. Sound type systems are useful *and* meaningful. A soundly typed program cannot go wrong, up to a well-defined set of run-time exceptions [29]. When a typed program raises an exception, the accompanying message usually pinpoints the location of the problem in the program source.

From this description it is clear why programmers eventually wish to annotate programs in untyped languages with types and, ideally, with sound types. Types directly and indirectly increase a programmer's productivity, and sound types help with testing, debugging, and other maintenance tasks. In short, sound gradual typing seems to be a panacea.

The problem is that, according to our measurements, the cost of enforcing soundness is overwhelming. Figures 4 and 5 clarify just 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 amount of time, meaning that L=2 is again a liberal choice. But even this liberal choice does not increase the number of acceptable configurations by much; worse, it unrealistically assumes those two modules best-suited to improve performance. Put differently, the number of L-step N/M-acceptable configurations remains small with liberal choices for all three parameters.

The application of our evaluation method projects an extremely negative image of *sound* gradual typing. While we are confident that the method captures the spirit of the goals of gradual typing, our particular application of the method and its results must be put in perspective. Section 5.1 explains why the evaluation of Typed Racket may look overly negative. Section 5.2 presents an analysis of the worst elements in the twelve lattices and highlights those kinds of contracts that impose the most significant cost.

5.1 Threats to Validity of Conclusion

We have identified four threats to validity. First, our benchmarks are relatively small due to constraints on our computing infrastructure, but even those consume considerable resources. To obtain results for these benchmarks in a reasonable amount of time, they are run using multiple cores and the configurations are divided amongst the cores. Each configuration is put into a single process running a separate instance of the Racket VM pinned to a single core. This parallelism may introduce confounding variables due to, e.g., 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. Regardless, given the low typed/untyped ratios, these libraries are unlikely to affect our conclusions.

Third, the feasible set of type annotations for a program component is rarely unique in a gradually typed system. Since types are translated into contracts in Typed Racket, the choice of type annotations may affect performance. All of our case studies use reasonable type annotations, but type annotations with superior performance may exist. For example, one class-based benchmark (not included, completed after submission) exhibits noticeable differences though the overall result remains the same. Generally speaking, our results may not be fully representative. Then again, it is still a failure of gradual typing if a programmer must divine the best possible type annotations to obtain reasonable performance.

Finally, we articulate our conclusions on the basis of current implementation technology. Typed Racket compiles to Racket, which uses rather conventional JIT 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.

5.2 What are the Bottlenecks?

To analyze the cost of contract checks, we used the feature-specific profiler [23] on each benchmark's *slowest* configuration.[9] Figure 6 summarizes our findings.

[9] We found no statistically significant difference in the proportion of runtimes spent in garbage collection between the untyped & slowest configurations of any benchmark.

The leftmost data column (%C) gives the percent of each benchmark's total running time that was spent checking contracts. These percentages are the average of ten trials; the numbers in parentheses (S.E.) represent the standard error. Except for the short-running benchmarks (`gregor`, `morse-code`, and `mbta`), we see little variability across trials. As expected, the programs spend a substantial proportion of their running time checking contracts.

The remaining columns of figure 6 report what percentage of each benchmark's *contract-checking* execution time is spent on a particular variety of contract:

- Adaptor contracts separate a typed module from an untyped module with data structures.
- Higher-order contracts are function contracts with at least one function in their domain or co-domain.
- Library contracts separate an untyped library from typed modules or vice versa (in the case of `lnm`).
- The shape `(-> T any/c)` refers to contracts with a protected argument and an unchecked co-domain. Contracts of this shape typically guard typed functions called in untyped modules.
- Conversely, `(-> any/c T)` guards functions with (any number of) unchecked arguments and protected co-domains. For example, if a typed module calls an untyped function with immutable arguments, Typed Racket statically proves that the untyped function is given well-typed arguments but must insert a contract to verify the function's result.
- The `(-> any/c boolean?)` column measures the time spent checking functions that take a single argument and returning a Boolean value. It is thus a subset of the `(-> any/c T)` column.

Other columns overlap as well. The `mbta` benchmark in particular spends 65% of its contract-checking time on first-order library functions. These checks are always triggered by a typed module on immutable arguments, so Typed Racket optimizes them to `(-> any/c T)` contracts.

Most strikingly, the `(-> any/c boolean?)` column suggests that on average twenty percent of the time our benchmarks spend checking contracts goes towards checking that predicate functions satisfy the trivial `(-> any/c boolean?)` contract. Moreover, nearly all of these predicates are generated by Racket structure definitions, so their type correctness might be assumed. Removing these contracts or optimizing the cost of indirection seems like a clear place for Typed Racket to improve.

In contrast, the adaptor and library columns suggest that the apparently high cost of predicate contracts may just be a symptom of placing a typed/untyped boundary between a structure type definition and functions closely associated with the data. One example of this is `zordoz`; indeed, the purpose of that code is to provide an interface to native compiler data structures. In nearly all worst-case measurements for benchmarks using adaptor modules the adaptor and `(-> any/c boolean?)` contracts seem to account for a huge proportion of all contracts. The `quad` benchmark in fact spends 93% of its contract-checking time validating data structures, which are stored in fixed-length lists rather than in structure types. These lists do not require an adaptor, but their types translate to contracts that are far more expensive than plain structure type predicates. The only exception is `synth`. It spends much more time creating structured data from raw vectors than accessing the data.

Higher-order contracts show up in only a few of the benchmark programs. Specifically, only `synth`, `sieve`, and `zordoz` make heavy use of higher-order functions across contract boundaries. Unlike the cost of first-order contracts, the costs of these higher-order contracts is quite apparent in these programs.

Project	%C	(S.E.)	adaptor	higher-order	library	(-> T any/c)	(-> any/c T)	(-> any/c boolean?)
sieve	92	(2.33)	0	46	0	0	54	31
morse-code	29	(6.8)	0	0	0	0	100	0
mbta	39	(3.65)	0	0	65	0	65	0
zordoz	95	(0.1)	0	55	45	0	99	43
suffixtree	94	(0.18)	98	<1	0	2	94	18
lnm	81	(0.73)	0	9	99	91	0	0
kcfa	91	(0.26)	100	0	0	0	54	31
snake	98	(0.21)	93	0	0	1	99	49
tetris	96	(0.35)	89	0	0	11	89	44
synth	83	(1.22)	51	90	0	29	20	0
gregor	83	(4.01)	78	0	3	7	85	31
quad	80	(0.96)	<1	1	0	3	<1	<1

Figure 6: Profiling the worst-case contract overhead

Finally, the `(-> T any/c)` and `(-> any/c T)` columns give a rough impression of whether untyped or typed modules trigger more contract checks. We confirmed these findings by inspecting the individual programs. For all but three benchmarks, the high-cost contracts are triggered by calls from a typed module into an untyped library or data definition. This includes `kcfa`, although half its calls from typed to untyped code used mutable arguments and hence could not be reduced to `any/c`. The exceptions are `lnm`, `synth`, and `quad`, which suffer from slowdowns when untyped modules import definitions from typed ones.

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 [28], Smalltalk [2], Thorn [7] and TypeScript [18, 19]. 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 [28, 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's evaluation is based on the TypeScript ports of the Octane benchmarks. Unlike our lattice-based approach, it compares only the performance of the fully untyped and fully typed programs. 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" [18, pg. 178]. On fully typed programs, the overhead is "on average only 6.5%" [18, 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 [7]. StrongScript follows Thorn's lead by adding a sound type system (with a limited form of higher-order wrappers) to TypeScript. Thorn has a minimal performance evaluation which shows that by sprinkling a few type annotations over toy benchmarks, speed-ups between 3x and 6x can be obtained [30]. 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 to 20% speedups) on the fully-typed versions [19, 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 can be traced as far back as MACLISP, which allowed users to declare (unchecked) type specifications [15, §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 [9, 10], and also belong to the unsound camp. Recent implementations, e.g. Papi et al.'s work for Java [17], layer additional typed reasoning on top of existing typed languages rather than untyped languages.

Contemporary optional type systems have been developed for Clojure [8], Lua [14], Python,[10] PHP,[11] ActionScript,[12] Dart,[13] and JavaScript [6]. Since the type annotations in these systems are unsound for typed-untyped interoperation, they incur no runtime overhead from proxy wrapping or dynamic checks. The lack of overheads obviates the need for a performance evaluation such as the one in this paper.

Some 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 to generate faster code. This intuition is borne out by Chang et al. [11] who report significant speed-ups for typed Ac-

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

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

[12] `http://help.adobe.com/en_US/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7f8a.html`

[13] `http://dartlang.org`

tionScript 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 the baseline. Richards et al. [19] 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 typing, and it supports only macro-level gradual typing. Before we declare gradual typing completely dead, we must apply our method to other implementations. The question is whether doing so will yield equally negative results. Safe TypeScript [18] appears to be one natural candidate for such an effort. At the same time, we are also challenged to explore how our evaluation method 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 untouched. 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 other 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 [27]. The latter implements a JIT compiler that open-codes primitive functions. 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. A sophisticated JIT compiler may eliminate some of the contract overhead in such cases, but we conjecture that performance pathologies will still remain. Applying our method to an implementation with a more sophisticated compiler, e.g., Pycket [5], may let us validate this conjecture.

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. Expanding 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 [23] and optimization coaches [24] 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 the use of our method will yield useful performance evaluations to guide future research. Above we have spelled out practical directions but even theoretical ideas—such as Henglein's optimal coercion insertion [13] and the collapsing of chains of contracts [22]—may take inspiration from the application of our method.

Data and Code

Our benchmarks and measurements are available in our artifact: `http://www.ccs.neu.edu/racket/pubs/#popl15-tfgnvf`

Acknowledgments

The authors gratefully acknowledge support from the National Science Foundation (SHF 1518844). They also thank Matthew Butterick, John Clements, Matthew Might, Vincent St-Amour, Neil Toronto, David Van Horn, Danny Yoo, and Jon Zeppieri for providing benchmark code bases. Brian LaChance and Sam Tobin-Hochstadt provided valuable feedback on earlier drafts.

References

[1] Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on 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* 96(1), pp. 52–69, 2013.

[3] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined Gradual Typing. In *Proc. ACM Conference on 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] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfield, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A Tracing JIT For a Functional Language. In *Proc. ACM International Conference on Functional Programming*, pp. 22–34, 2015.

[6] Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *Proc. European Conference on Object-Oriented Programming*, pp. 257–281, 2014.

[7] 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 Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 117–136, 2009.

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

[9] Gilad Bracha. Pluggable Type Systems. In *Proc. OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

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

[11] 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.

[12] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM International Conference on Functional Programming*, pp. 48–59, 2002.

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

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

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

[16] Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification. In *Proc. ACM International Conference on Functional Programming*, pp. 139–152, 2014.

[17] Matthew M. Papi, Mahmood Ali, Telmo Louis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical Pluggable Types for Java. In *Proc. International Symposium on Software Testing and Analysis*, pp. 201–212, 2008.

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

[19] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *Proc. European Conference on Object-Oriented Programming*, pp. 76–100, 2015.

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

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

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

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

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

[25] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. In *Proc. European Conference on Object-Oriented Programming*, pp. 4–27, 2015.

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

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

[28] 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.

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

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