Under consideration for publication in J. Functional Programming

How to Evaluate the Performance of Gradual Type Systems

BEN GREENMAN[1] ASUMU TAKIKAWA[1,3] MAX S. NEW[1] DANIEL FELTEY[1,2]
ROBERT BRUCE FINDLER[2] JAN VITEK[1,4]and MATTHIAS FELLEISEN[1]
Northeastern University, Boston, Mass.[1] Northwestern University, Chicago, Ill.[2]
Igalia, San Francisco, Cal.[3] Czech Technical University, Prague, CZ [4]

Abstract

A sound gradual type system ensures that untyped components of a program can never break the guarantees of statically typed components. This assurance relies on runtime checks, which in turn impose performance overhead in proportion to the frequency and nature of interaction between typed and untyped components.

The literature on gradual typing lacks rigorous descriptions of methods for measuring the performance of gradual type systems. This gap has consequences for the implementors of gradual type systems and developers who use such systems. Without systematic evaluation of mixed-typed programs, implementors cannot precisely determine how improvements to a gradual type system affect performance. Developers cannot predict whether adding types to part of a program will significantly degrade (or improve) its performance.

This paper presents the first method for evaluating the performance of sound gradual type systems. The method quantifies both the absolute performance of a gradual type system and the relative performance of two implementations of the same gradual type system. To validate the method, the paper reports on its application to twenty programs and three implementations of Typed Racket.

1 The Gradual Typing Design Space

Programmers use dynamically typed languages to build all kinds of applications. Telecom companies have been running Erlang programs for years; Sweden's pension system is a Perl program (Lemonnier 2006); web developers build large programs with JavaScript and Node.js; and the server-side applications of Dropbox, Facebook, and Twitter were originally written in Python, PHP, and Ruby, respectively.

Regardless of why programmers choose dynamically typed languages, the maintainers of these applciations often face a problem; namely, the lack of explicit and reliable type annotations poses an obstacle to their work. Researchers have tried to overcome the lack of type annotations with type inference for at least two decades (Aiken et al. 1994; Anderson et al. 2005; Cartwright and Fagan 1991; Flanagan et al. 1996; Furr et al. 2009; Henglein and Rehof 1995; Suzuki 1981), but most have come to realize that there is no substitute for programmer-supplied annotations. Unlike types inferred from the structure of the code, explicit annotations communicate the original programmer's specification to other human readers and thereby serve as a guide when it comes time to repair or extend the code.

One approach to the maintenance problem is to rewrite the entire application in a statically typed language. For example, engineers at Twitter ported their server-side code from Ruby to Scala because they decided the long-term performance and readability benefits outweighed the cost of the rewrite.[1]

Another approach is gradual typing (Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006).[2] In a sound gradually typed language, programmers can incrementally add type annotations to dynamically typed code. At the lexical boundaries between annotated code and dynamically typed code, the type system inserts runtime checks to guarantee type soundness (Tobin-Hochstadt et al. 2017), which ensures that no typed operation is applied to arguments outside its domain.

From a syntactic perspective the interaction between annotated and unannotated code is seamless, but dynamic checks introduce runtime overhead. If, during execution, an untyped function flows into a variable f with type $\tau_1 \rightarrow \tau_2$, then a dynamic check must guard every subsequent call to f because typed code cannot assume that values produced by the function have the syntactic type τ_2. Conversely, typed functions invoked by dynamically-typed code must check their argument values. If functions or large data structures frequently cross these *type boundaries*, enforcing type soundness might impose a significant runtime cost and thus make the entire approach impractical.

Optimistically, researchers have continued to explore the theory of sound gradual typing (Garcia and Cimini 2015; Garcia et al. 2016; Knowles et al. 2007; Siek and Vachharajani 2008; Takikawa et al. 2012; Wolff et al. 2011).[3] Some research groups have invested significant resources implementing sound gradual type systems (Allende et al. 2013; Rastogi et al. 2015; Tobin-Hochstadt et al. 2017; Vitousek et al. 2017). But surprisingly few groups have evaluated the performance of gradual typing. Many acknowledge an issue with performance in passing (Allende et al. 2013; Tobin-Hochstadt and Felleisen 2008; Vitousek et al. 2014). Others report only the performance of fully annotated programs relative to fully unannotated programs (Rastogi et al. 2015; Vitousek et al. 2017), ignoring the entire space of programs that mix typed and untyped components.

This archival paper presents the first method for evaluating the performance of a gradual type system (section 3), integrating new results with a conference version (Takikawa et al. 2016). Specifically, this paper contributes:

- validation that the method can effectively compare the performance of three implementations of the same gradual type system (section 5.3);
- evidence that simple random sampling can approximate the results of an exhaustive evaluation with asymptotically fewer measurements (section 6);
- eight object-oriented benchmark programs to augment the functional benchmarks of Takikawa et al. (2016) (section 4); and
- a discussion of the pathological overheads in the benchmark programs (section 8).

The discussion in section 8 is intended to guide implementors of gradual type systems toward promising future directions (section 9). In particular, there are challenges to ad-

[1] `http://www.artima.com/scalazine/articles/twitter_on_scala.html`
[2] Tobin-Hochstadt et al. (2017) refer to this use of gradual typing as *migratory typing*.
[3] See `https://github.com/samth/gradual-typing-bib` for a bibliography.

guess-game

```
#lang racket

(provide play)

(define (play)
  (define n (random 10))
  (λ (guess)
    (= guess n)))
```

player

```
#lang racket

(provide stubborn-player)

(define (stubborn-player i)
  4)
```

driver

```
#lang typed/racket

(require/typed "guess-game.rkt"
  [play (-> (-> Natural Boolean))])
(require/typed "player.rkt"
  [stubborn-player (-> Natural Natural)])

(define check-guess (play))
(for/sum ([i : Natural (in-range 10)])
  (if (check-guess (stubborn-player i)) 1 0))
```

Figure 1: A gradually typed application

dress both in the implementation of sound gradual type systems and in helping users find combinations of typed and untyped code that meet their performance requirements.

2 Gradual Typing in Typed Racket

Typed Racket (Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt et al. 2017) is a sound gradual type system for Racket. Its syntax is an extension of Racket's; its static type checker supports idioms common in dynamically-typed Racket programs (Strickland et al. 2009; Takikawa et al. 2015); and a Typed Racket module may import definitions from a Racket module. A Racket module may likewise import definitions from a Typed Racket module. To ensure type soundness, Typed Racket compiles static types to higher-order contracts and attaches these contracts at the lexical boundaries between Typed Racket and Racket modules (Tobin-Hochstadt et al. 2017).

Figure 1 demonstrates gradual typing in Typed Racket with a toy application. The module on the top left implements a guessing game with the function `play`. Each call to `play` generates a random number and returns a function that compares a given number to this chosen number. The module on the top right implements a simple player. The driver module at the bottom combines the game and player. It instantiates the game, prompts `stubborn-player` for ten guesses, and counts the number of correct guesses using the `for/sum` comprehension. Of the three modules, only the driver is implemented in Typed Racket. This means that Typed Racket statically checks the body of the driver module under the assumption that its annotations for the `play` and `stubborn-player` functions are correct. Typed Racket guarantees this assumption about these functions by compiling their type annotations into dynamically-checked contracts. One contract checks that `(play)` returns a function from natural numbers to booleans (by attaching a new contract to the returned value), and the other checks that `stubborn-player` returns natural numbers.

2.1 How Mixed-Typed Code Emerges

The close integration of Racket and Typed Racket makes it easy to migrate a code base from the former to the latter on an incremental basis. By converting modules to Typed Racket, the maintainer obtains several benefits:

- *assurance* from the typechecker against basic logical errors;
- *documentation* in the form of reliable type annotations;
- *protection* against dynamically-typed clients; and
- *speed*, when the compiler can use types to improve the generated bytecode.

These benefits draw numerous Racket programmers to Typed Racket.

Another way that Racket programs may rely on Typed Racket is by importing definitions from a typed library. For example, the creator of Racket's `plot` library wrote most of the library in Typed Racket, and thus the program that typeset this paper interacts with typed code. This kind of import is indistinguishable from importing definitions from a Racket library and is a rather subtle way of creating a mixed-typed codebase.

Conversely, even developers who start with Typed Racket programs rely on Racket in the form of legacy libraries. For such libraries, the programmer has a choice between writing a type-annotated import statement for the library API and converting the entire library to Typed Racket. Most choose the former approach.

In summary, the introduction of Typed Racket has given rise to a fair number of projects that mix typed and untyped code.[4] The amount of mixed-typed code in the ecosystem is likely to grow; therefore, it is essential that Typed Racket programs interact smoothly with existing Racket code.

2.2 The Need for Performance Evaluation

Typed Racket is an evolving research project. The initial release delivered a sound type system that could express the idioms in common Racket programs. Subsequent improvements focused on growing the type system to support variable-arity polymorphism (Strickland et al. 2009), control operators (Takikawa et al. 2013), and the object-oriented extensions to Racket (Takikawa et al. 2015). All along, the Typed Racket developers knew that the performance cost of enforcing type soundness could be high (see section 8 for rationale); however, this cost was not an issue in many of their programs.

As other programmers began using Typed Racket, a few discovered issues with its performance. For instance, one user experienced a 1.5x slowdown in a web server, others found 25x–50x slowdowns when using an array library, and two others reported over 1000x slowdowns when using a library of functional data structures.[5] Programmers found these performance issues unacceptable, because the slowdowns were large, difficult to predict,

[4] See `pkgd.racket-lang.org/pkgn/search?q=typed-racket` for a sample.
[5] The appendix samples these user reports.

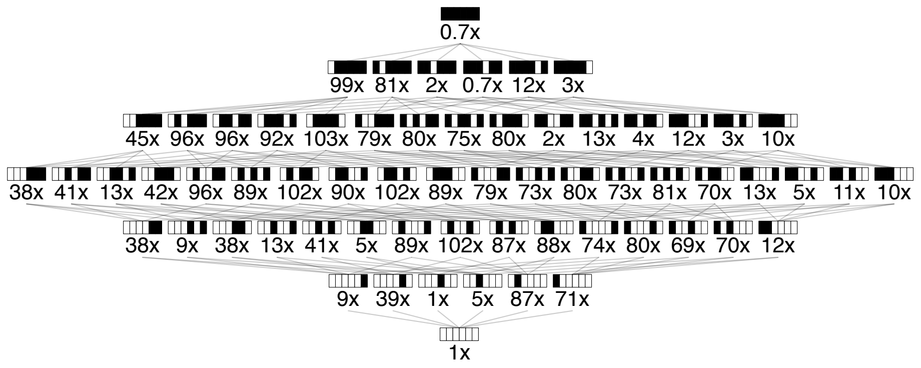

0.7x
99x 81x 2x 0.7x 12x 3x
45x 96x 96x 92x 103x 79x 80x 75x 80x 2x 13x 4x 12x 3x 10x
38x 41x 13x 42x 96x 89x 102x 90x 102x 89x 79x 73x 80x 73x 81x 70x 13x 5x 11x 10x
38x 9x 38x 13x 41x 5x 89x 102x 87x 88x 74x 80x 69x 70x 12x
9x 39x 1x 5x 87x 71x
1x

Figure 2: Performance overhead in `suffixtree`, on Racket v6.2.

and difficult to debug. Instead of making a program run more efficiently, adding types to the wrong boundary seriously degraded performance, negating an advertised benefit. In short, programmers' experience with Typed Racket forced the Typed Racket team to develop a systematic method of performance evaluation.

3 Evaluation Method, Part I

A performance evaluation of gradual type systems must reflect how programmers use such systems. Since experience with Typed Racket shows that programmers frequently combine typed and untyped code within an application, an evaluation method must account for the large space of code between untyped and fully-typed programs. These applications may also undergo gradual transitions that add or remove some type annotations. In a typical evolution, programmers compare the performance of the modified program with the previous version. If type-driven optimizations result in a performance improvement, all is well. Otherwise, the programmer may need to address the performance overhead. As they continue to develop the application, programmers repeat this process. Again, an evaluation method must take into account gradual evolution to reflect practice.

The following subsections build an evaluation method from these observations in three steps. First, section 3.1 describes the space over which a performance evaluation must take place. Second, section 3.2 defines metrics relevant to the performance of a gradually typed program. Third, section 3.3 introduces a visualization that concisely presents the metrics on exponentially large spaces.

3.1 Performance Lattice

The promise of Typed Racket's macro-level gradual typing is that programmers can add types to any subset of the modules in an untyped program. In principle, this promise extends to third-party libraries and modules from the Racket runtime system, but in practice a programmer has no control over such modules. Thus we distinguish between two kinds of modules: the *migratable* modules that a programmer may add types to, and the *contextual* modules in the software ecosystem, which remain unchanged. A comprehensive performance evaluation must therefore consider the *configurations* a programmer could possibly create given type annotations for each migratable module. These configurations form a lattice, ordered by the subset relation on the set of typed modules in a configuration.

Figure 2 demonstrates one such lattice for a program with six migratable modules. The black rectangle at the top of the lattice represents the configuration in which all six modules are typed. The other 63 rectangles represent configurations with some untyped modules.

A given row in the lattice groups configurations with the same number of typed modules (black squares). For instance, configurations in the second row from the bottom contain two typed modules. These represent all possible ways of converting two modules in the untyped configuration to Typed Racket. Similarly, configurations in the third row represent all possible configurations a programmer might encounter after applying three such *type conversion steps* to the untyped configuration. In general, let the notation $c_1 \rightarrow_k c_2$ express the idea that a programmer starting from configuration c_1 (in row i) could reach configuration c_2 (in row j) after taking at most k type conversion steps ($j - i \leqslant k$).

Configurations in figure 2 are furthermore labeled with their performance overhead relative to the untyped configuration on Racket version 6.2. With these labels, a language implementor can draw several conclusions about the performance overhead of gradual typing in this program. For instance, eight configurations run within a 2x overhead and 22 configurations are at most one type conversion step from a configuration that runs within a 2x overhead. High overheads are common (38 configurations have over 20x overhead), but the fully typed configuration runs faster (0.7x overhead) than the untyped configuration because Typed Racket uses the type annotations to compile efficient bytecode.

A labeled lattice such as figure 2 is a *performance lattice*. The same lattice without labels is a *configuration lattice*. Practically speaking, users of a gradual type system explore configuration lattices and maintainers of such systems may use performance lattices to evaluate overall performance.

3.2 Performance Metrics

The most basic question, and least important, about a gradually typed language is how fast fully-typed programs are in comparison to their fully untyped relative. In principle and in Typed Racket, static types enable optimizations and can serve in place of runtime tag checks. The net effect of such improvements may, however, be offset by the runtime cost of enforcing type soundness. Relative performance is therefore best described as a ratio, to capture the possibility of speedups and slowdowns.

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

For users of a gradual type system, the important performance question is how much overhead their current configuration suffers. If the performance overhead is low enough, programmers can release the configuration to clients. Depending on the nature of the application, some developers might not accept any performance overhead. Others may be willing to tolerate an order-of-magnitude slowdown. The following parameterized definition of a deliverable configuration accounts for these varying requirements.

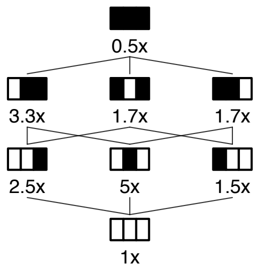


Figure 3: Sample performance lattice

Definition (*D-deliverable*) A configuration is *D*-deliverable if its performance is no worse than a factor of *D* slowdown compared to the untyped configuration.

If an application is currently in a non-*D*-deliverable configuration, the next question is how much work a team must invest to reach a *D*-deliverable configuration. One coarse measure of "work" is the number of additional modules that must be annotated with types before performance improves.

Definition (*k-step D-deliverable*) A configuration c_1 is *k*-step *D*-deliverable if $c_1 \rightarrow_k c_2$ and c_2 is *D*-deliverable.

The number of *k*-step *D*-deliverable configurations captures the experience of a prescient programmer that converts the *k* modules that maximize the performance improvement.

Let us illustrate these terms with an example. Suppose there is a project with three modules where the untyped configuration runs in 20 seconds and the typed configuration runs in 10 seconds. Furthermore, suppose half the mixed configurations run in approximately 50 seconds and the other half run in approximately 30 seconds. Figure 3 is a performance lattice for this hypothetical program. The label below each configuration is its overhead relative to the untyped configuration.

The typed/untyped ratio is 1/2, indicating a performance improvement due to adding types. The typed configuration is also 1-deliverable because it runs within a 1x slowdown relative to the untyped configuration. All mixed configurations are 5-deliverable, but only three are, e.g., 2-deliverable. Lastly, the mixed configurations are all 2-step 1-deliverable because they can reach the typed configuration in at most two type conversion steps.

The ratio of *D*-deliverable configurations in a performance lattice is a measure of the overall feasibility of gradual typing. When this ratio is high, then no matter how the application evolves, performance is likely to remain acceptable. Conversely, a low ratio implies that a team may struggle to recover performance after typing a few modules. Practitioners with a fixed performance requirement *D* can therefore use the number of *D*-deliverable configurations to extrapolate the performance of a gradual type system.

3.3 Overhead Plots

Although a performance lattice contains a comprehensive description of performance overhead, it does not effectively communicate this information. It is difficult to tell, at a glance,

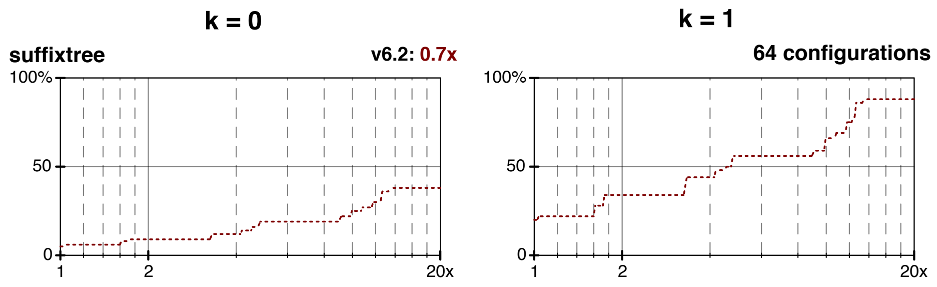


Figure 4: Overhead plots for `suffixtree`, on Racket v6.2. The unlabeled vertical ticks mark, from left-to-right: 1.2x, 1.4x, 1.6x, 1.8x, 4x, 6x, 8x, 10x, 12x, 14x, 16x, and 18x.

whether a program has good or bad performance relative to its users' requirements. Comparing the relative performance of two or more lattices is also difficult and is in practice limited to programs with an extremely small number of modules.

The main lesson to extract from a performance lattice is the proportion of k-step D-deliverable configurations for various k and D. In other words, this proportion describes the number of configurations (out of the entire lattice) that are at most k upward steps from a D-deliverable configuration. One way to plot this information is to fix a value for k, say k=0, and consider a set of values $d_0,\ldots,d_{n-1}$ for D. The set of proportions of 0-step d_i-deliverable configurations defines a cumulative distribution function with the value of D on the independent axis and the proportion of configurations on the dependent axis.

Figure 4 demonstrates two such *overhead plots*, summarizing the data in figure 2. Specifically, each plots the 64 configurations of a program called `suffixtree` using data measured on Racket v6.2. The plot on the left fixes k=0 and plots the proportion of 0-step D-deliverable configurations. The plot on the right fixes k=1 and plots the proportion of 1-step D-deliverable configurations. Both plots consider 250 values of D evenly spaced between 1x and 20x. The line on each plot represents the cumulative distribution function. The x-axis is log scaled to focus on low overheads.

The plot on the left, in which k=0, confirms the observation made in section 3.1 that 12% of the 64 configurations (eight configurations) run within a 2x overhead. For values of D larger than 2x, the proportion of D-deliverable configurations is slightly larger, but even at a 20x overhead, this proportion is only 41%. The plot on the right shows that the proportion of 1-step D-deliverable is typically twice as high as the proportion of D-deliverable configurations for this benchmark.

This presentation scales to arbitrarily large programs because the y-axis plots the proportion of D-deliverable configurations; in contrast, a performance lattice contains exponentially many nodes. Furthermore, plotting the overhead for multiple implementations of a gradual type system on the same set of axes conveys their relative performance.

3.3.1 *Assumptions and Limitations*

Plots in the style of figure 4 rest on two assumptions and have two significant limitations, which readers must keep in mind as they interpret the results.

The *first assumption* is that configurations with less than 2x overhead are significantly more practical than configurations with a 10x overhead or more. Hence the plots use a log-scaled x-axis to simultaneously encourage fine-grained comparison in the 1.2x to 1.6x overhead range and blur the distinction between, e.g., 14x and 18x slowdowns.

The *second assumption* is that configurations with more than 20x overhead are completely unusable in practice. Pathologies like the 100x slowdowns in figure 2 represent a challenge for implementors, but if these overheads suddenly dropped to 30x, the configurations would still be useless to developers.

The *first limitation* of the overhead plots is that they do not report the number of types in a configuration. The one exception is the fully-typed configuration; its overhead is given explicitly through the typed/untyped ratio above the left plot.

The *second limitation* is that the 1-step *D*-deliverable plot does not show how we optimistically chose the best type conversion step. In a program with *N* modules, a programmer has at most *N* type conversion steps to choose from, some of which may not lead to a *D*-deliverable configuration. For example, there are six configurations with exactly one typed module in figure 2 but only one of these is 1-deliverable.

4 The GTP Benchmark Programs

This section introduces twenty *gradual typing performance* (GTP) benchmark programs that are representative of actual user code yet small enough to make exhaustive performance evaluation tractable. The following descriptions of the benchmark programs—arranged by size as measured in number of migratable modules—briefly summarize their relevant characteristics. Each description comes with four fields: *Origin* indicates the benchmark's source, *Purpose* describes what it computes, *Author* credits the original author, and *Depends* lists any libraries of contextual modules that the benchmark depends on. This section concludes with a table summarizing the static characteristics of each benchmark.

sieve

Origin	: Synthetic	*Author*	: Ben Greenman
Purpose	: Generate prime numbers	*Depends*	: N/A

Demonstrates a scenario where client code is tightly coupled to higher-order library code. The library implements a stream data structure; the client builds a stream of prime numbers. Introducing a type boundary between these modules leads to significant overhead.

forth

Origin	: Library	*Author*	: Ben Greenman
Purpose	: Forth interpreter	*Depends*	: N/A

Interprets Forth programs. The interpreter represents calculator commands as a list of first-class objects. These objects accumulate proxies as they cross type boundaries.

fsm, fsmoo

Origin	: Economics Research	*Author*	: Linh Chi Nguyen
Purpose	: Economy Simulator	*Depends*	: N/A

Simulates the interactions of economic agents via finite-state automata (Nguyen 2014). This benchmark comes in two flavors: `fsm` stores the agents in a mutable vector and whereas `fsmoo` uses a first-class object.

mbta

Origin	: Educational	*Author*	: Matthias Felleisen
Purpose	: Interactive map	*Depends*	: `graph`

Builds a map of Boston's subway system and answers reachability queries. The map encapsulates a boundary to Racket's untyped `graph` library; when the map is typed, the (type) boundary to `graph` is a performance bottleneck.

morsecode

Origin	: Library	*Author*	: John Clements and Neil Van Dyke
Purpose	: Morse code trainer	*Depends*	: N/A

Computes Levenshtein distances and morse code translations for a fixed sequence of pairs of words. Every function that crosses a type boundary in `morsecode` operates on strings and integers, thus dynamically type-checking these functions' arguments is relatively cheap.

zombie

Origin	: Research	*Author*	: David Van Horn
Purpose	: Game	*Depends*	: N/A

Implements a game where players dodge computer-controlled "zombie" tokens. Curried functions over symbols implement game entities and repeatedly cross type boundaries.

dungeon

Origin	: Application	*Author*	: Vincent St. Amour
Purpose	: Maze generator	*Depends*	: N/A

Builds a grid of wall and floor objects by choosing first-class classes from a list of "template" pieces. This list accumulates proxies when it crosses a type boundary.

zordoz

Origin	: Library	*Author*	: Ben Greenman
Purpose	: Explore Racket bytecode	*Depends*	: `compiler-lib`

Traverses Racket bytecode (`.zo` files). The `compiler-lib` library defines the bytecode data structures. Typed code interacting with the library suffers overhead.

Note: the Racket bytecode format changed between versions 6.2 and 6.3 with the release of the set-of-scopes macro expander (Flatt 2016). This change significantly reduced the overhead of `zordoz`.

lnm

Origin	: Synthetic	*Author*	: Ben Greenman
Purpose	: Graphing	*Depends*	: `plot`, `math/statistics`

Renders overhead plots (Takikawa et al. 2016). Two modules are tightly-coupled to Typed Racket libraries; typing both modules improves performance.

suffixtree

Origin : Library		*Author* : Danny Yoo	
Purpose : Ukkonen's suffix tree algorithm		*Depends* : N/A	

Computes longest common subsequences between strings. The largest performance overheads are due to a boundary between struct definitions and functions on the structures.

kcfa

Origin : Educational	*Author* : Matt Might
Purpose : Explanation of k-CFA	*Depends* : N/A

Performs 1-CFA on a lambda calculus term that computes `2*(1+3) = 2*1 + 2*3` via Church numerals. The (mutable) binding environment flows throughout functions in the benchmark. When this environment crosses a type boundary, it acquires a new proxy.

snake

Origin : Research	*Author* : David Van Horn
Purpose : Game	*Depends* : N/A

Implements the Snake game; the benchmark replays a fixed sequence of moves. Modules in this benchmark frequently exchange first-order values, such as lists and integers.

take5

Origin : Educational	*Author* : Matthias Felleisen
Purpose : Game	*Depends* : N/A

Manages a card game between AI players. These players communicate infrequently, so gradual typing adds relatively little overhead.

acquire

Origin : Educational	*Author* : Matthias Felleisen
Purpose : Game	*Depends* : N/A

Simulates a board game via message-passing objects. These objects encapsulate the core data structures; few higher-order values cross type boundaries.

tetris

Origin : Research	*Author* : David Van Horn
Purpose : Game	*Depends* : N/A

Replays a pre-recorded game of Tetris. Frequent type boundary crossings, rather than proxies or expensive runtime checks, are the source of performance overhead.

synth

Origin : Application	*Author* : Vincent St. Amour & Neil Toronto
Purpose : Music synthesis DSL	*Depends* : N/A

Converts a description of notes and drum beats to `WAV` format. Modules in the benchmark come from two sources, a music library and an array library. The worst overhead occurs because arrays frequently cross type boundaries.

gregor

Origin	: Library	*Author*	: Jon Zeppieri
Purpose	: Date and time library	*Depends*	: `cldr`, `tzinfo`

Provides tools for manipulating calendar dates. The benchmark builds tens of date values and runs unit tests on these values.

quadBG, quadMB

Origin	: Application	*Author*	: Matthew Butterick
Purpose	: Typesetting	*Depends*	: `csp`

Converts S-expression source code to `PDF` format. The two versions of this benchmark differ in their type annotations, but have nearly identical source code.

The original version, `quadMB`, uses type annotations by the original author. This version has a high typed/untyped ratio because it explicitly compiles types to runtime predicates and uses these predicates to eagerly check data invariants. In other words, the typed configuration is slower than the untyped configuration because it performs more work.

The second version, `quadBG`, uses identical code but weakens types to match the untyped configuration. This version is therefore suitable for judging the implementation of Typed Racket rather than the user experience of Typed Racket. The conference version of this paper included data only for `quadMB`.

Figure 5 tabulates the size and complexity of the benchmark programs. The lines of code (Untyped LOC) and number of migratable modules (# Mod.) approximate program size. The type annotations (Annotation LOC) count additional lines in the typed configuration. These lines are primarily type annotations, but also include type casts and assertions.[6] Adaptor modules (# Adp.) roughly correspond to the number of user-defined datatypes in each benchmark; the next section provides a precise explanation. Lastly, the boundaries (# Bnd.) and exports (# Exp.) distill each benchmark's graph structure.[7] Boundaries are import statements from one module to another, excluding imports from runtime or third-party libraries. An identifier named in such an import statement counts as an export. For example, the one import statement in `sieve` names nine identifiers.

4.1 From Programs to Benchmarks

Seventeen of the benchmark programs are adaptations of untyped programs. The other three benchmarks (`fsm`, `synth`, and `quad`) use most of the type annotations and code from originally-typed programs. Any differences between the original programs and the benchmarks are due to the following five complications.

First, the addition of types to untyped code occasionally requires type casts or small refactorings. For example, the expression `(string->number "42")` has the Typed Racket type `(U Complex #f)`. This expression cannot appear in a context expecting an `Integer` without an explicit type cast. As another example, the `quad` programs call a library function to partition a `(Listof (U A B))` into a `(Listof A)` and a

[6] The benchmarks use more annotations than Typed Racket requires because they give full type signatures for each import. Only imports from untyped modules require annotation.

[7] The appendix contains actual module dependence graphs.

Benchmark	Untyped LOC	Annotation LOC	# Mod.	# Adp.	# Bnd.	# Exp.
sieve	35	17 (49%)	2	0	1	9
forth	255	30 (12%)	4	0	4	10
fsm	182	56 (31%)	4	1	5	17
fsmoo	194	83 (43%)	4	1	4	9
mbta	266	71 (27%)	4	0	3	8
morsecode	159	38 (24%)	4	0	3	15
zombie	302	27 (9%)	4	1	3	15
dungeon	534	68 (13%)	5	0	5	37
zordoz	1378	215 (16%)	5	0	6	11
lnm	488	114 (23%)	6	0	8	28
suffixtree	537	129 (24%)	6	1	11	69
kcfa	229	53 (23%)	7	4	17	62
snake	160	51 (32%)	8	1	16	31
take5	327	27 (8%)	8	1	14	25
acquire	1654	304 (18%)	9	3	26	106
tetris	246	107 (43%)	9	1	23	58
synth	835	139 (17%)	10	1	26	51
gregor	945	175 (19%)	13	2	42	142
quadBG	6780	221 (3%)	14	2	27	160
quadMB	6706	294 (4%)	16	2	29	174

Figure 5: Static characteristics of the GTP benchmarks

`(Listof B)` using a predicate for values of type `A`. Typed Racket cannot currently prove that values which fail the predicate have type `B`, so the `quad` benchmarks replace the call with two filtering passes.

Second, Typed Racket cannot enforce certain types across a type boundary. For example, the core datatypes in the `synth` benchmark are monomorphic because Typed Racket cannot dynamically enforce parametric polymorphism on instances of an untyped structure.

Third, any contracts present in the untyped programs are represented as type annotations and in-line assertions in the derived benchmarks. The `acquire` program in particular uses contracts to ensure that certain lists are sorted and have unique elements. The benchmark enforces these conditions with explicit pre and post-conditions on the relevant functions.

Fourth, each *static import* of an untyped struct type into typed code generates a unique datatype. Typed modules that share instances of an untyped struct must therefore reference a common static import site. The benchmarks include additional contextual modules, called *adaptor modules*, to provide this canonical import site; for each module *M* in the original program that exports a struct, the benchmark includes an adaptor module that provides type annotations for every identifier exported by *M*. Adaptor modules add a layer of indirection, but this indirection does not add measurable performance overhead.

Fifth, some benchmarks use a different modularization than the original program. The `kcfa` benchmark is modularized according to comments in the original, single-module program. The `suffixtree`, `synth`, and `gregor` benchmarks each have a single file containing all their data structure definitions; the original programs defined these structures in the same module as the functions on the structures. Lastly, the `quadBG` benchmark has two fewer modules than `quadMB` because it inlines the definitions of two (large) data structures that `quadMB` keeps in separate files. Removing these boundaries has a negligible affect on performance overhead and greatly reduces the number of configurations.

5 Evaluating Typed Racket

5.1 Experimental Protocol

Sections 5.2 and 5.3 present the results of an exhaustive performance evaluation of the twenty benchmark programs on three versions of Racket. The data is the result of applying the following protocol for each benchmark and each version of Typed Racket:

- Select a random permutation of the configurations in the benchmark.[8]
- For each configuration: recompile, run once ignoring the result to control against JIT warmup, and run once more and record the running time. Use the standard Racket bytecode compiler, JIT compiler, and runtime settings.
- Repeat the above steps $N \geqslant 10$ times to produce a sequence of N running times for each configuration.
- Summarize each configuration with the mean of the corresponding running times.

Specifically, a Racket script implementing the above protocol collected the data in this paper. The script ran on a dedicated Linux machine; this machine has two physical AMD Opteron 6376 processors (with 16 cores each) and 128GB RAM.[9] For the `quadBG` and `quadMB` benchmarks, the script utilized 30 of the machine's physical cores to collect data in parallel.[10] For all other benchmarks, the script utilized only two physical cores. Each core ran at minimum frequency as determined by the `powersave` CPU governor (approximately 1.40 GHz).

The online supplement to this paper contains both the experimental scripts and the full datasets. Section 7 reports threats to validity regarding the experimental protocol. The appendix discusses the stability of individual measurements and reports the number of running times for each configuration and version of Racket.

5.2 Evaluating Absolute Performance

Figures 6, 7, 8, and 9 present the results of measuring the benchmark programs in a series of overhead plots. As in figure 4, the left column of the figures are cumulative distribution functions for D-deliverable configurations and the right column are cumulative distribution functions for 1-step D-deliverable configurations. These plots include data for three versions of Racket released between June 2015 and February 2016. Data for version 6.2 are thin red curves with short dashes. Data for version 6.3 are mid-sized green curves

[8] In principle, there is no need to randomize the order, but doing so helps control against possible confounding variables (Mytkowicz et al. 2009).

[9] The Opteron is a NUMA architecture and uses the `x86_64` instruction set.

[10] Specifically, the script invoked 30 OS processes, pinned each to a CPU core using the `taskset` Linux command, waited for each process to report a running time, and collected the results.

with long dashes. Data for version 6.4 are thick, solid, blue curves. The typed/untyped ratio for each version appears above each plot in the left column.

Many curves are quite flat; they demonstrate that gradual typing introduces large and widespread performance overhead in the corresponding benchmarks. Among benchmarks with fewer than six modules, the most common shape is a flat line near the 50% mark. Such lines imply that the performance of a group of configurations is dominated by a single type boundary. Benchmarks with six or more modules generally have smoother slopes, but five such benchmarks have essentially flat curves. The overall message is that for many values of D between 1 and 20, few configurations are D-deliverable.

For example, in fourteen of the twenty benchmark programs, at most half the configurations are 2-deliverable on any version. The situation is worse for lower (more realistic) overheads, and does not improve much for higher overheads. Similarly, there are ten benchmarks in which at most half the configurations are 10-deliverable.

The curves' endpoints describe the extremes of gradual typing. The left endpoint gives the percentage of configurations that run at least as quickly as the untyped configuration. Except for the `lnm` benchmark, such configurations are a low proportion of the total.[11] The right endpoint shows how many configurations suffer over 20x performance overhead.[12] Nine benchmarks have at least one such configuration.

Moving from k=0 to k=1 in a fixed version of Racket does little to improve the number of D-deliverable configurations. Given the slopes of the k=0 plots, this result is not surprising. One type conversion step can eliminate a pathological boundary, such as those in `fsm` and `zombie`, but the overhead in larger benchmarks comes from a variety of type boundaries. Except in configurations with many typed modules, adding types to one additional module is not likely to improve performance.

In summary, the application of the evaluation method projects a negative image of Typed Racket's sound gradual typing. Only a small number of configurations in the benchmark suite run with low overhead; a mere 2% of all configurations are 1.4-deliverable on Racket v6.4. Many demonstrate extreme overhead; 66% of all configurations are not even 20-deliverable on version 6.4.

5.3 Evaluating Relative Performance

Although the absolute performance of Racket version 6.4 is underwhelming, it is a significant improvement over versions 6.2 and 6.3. This improvement is manifest in the difference between curves on the overhead plots. For example in `gregor` (third plot in figure 9), version 6.4 has at least as many deliverable configurations as version 6.2 for any overhead on the x-axis. The difference is greatest near x=2; in terms of configurations, over 80% of `gregor` configurations are not 2-deliverable on v6.2 but are 2-deliverable on v6.4. The overhead plots for many other benchmarks demonstrate a positive difference between the number of D-deliverable configurations on version 6.4 relative to version 6.2.

[11] The `sieve` case is degenerate. Only its untyped and fully-typed configurations are 1-deliverable.

[12] Half the configurations for `dungeon` do not run on versions 6.2 and 6.3 due to a defect in the way these versions proxy first-class classes. The overhead plots report an "over 20x" performance overhead for these configurations.

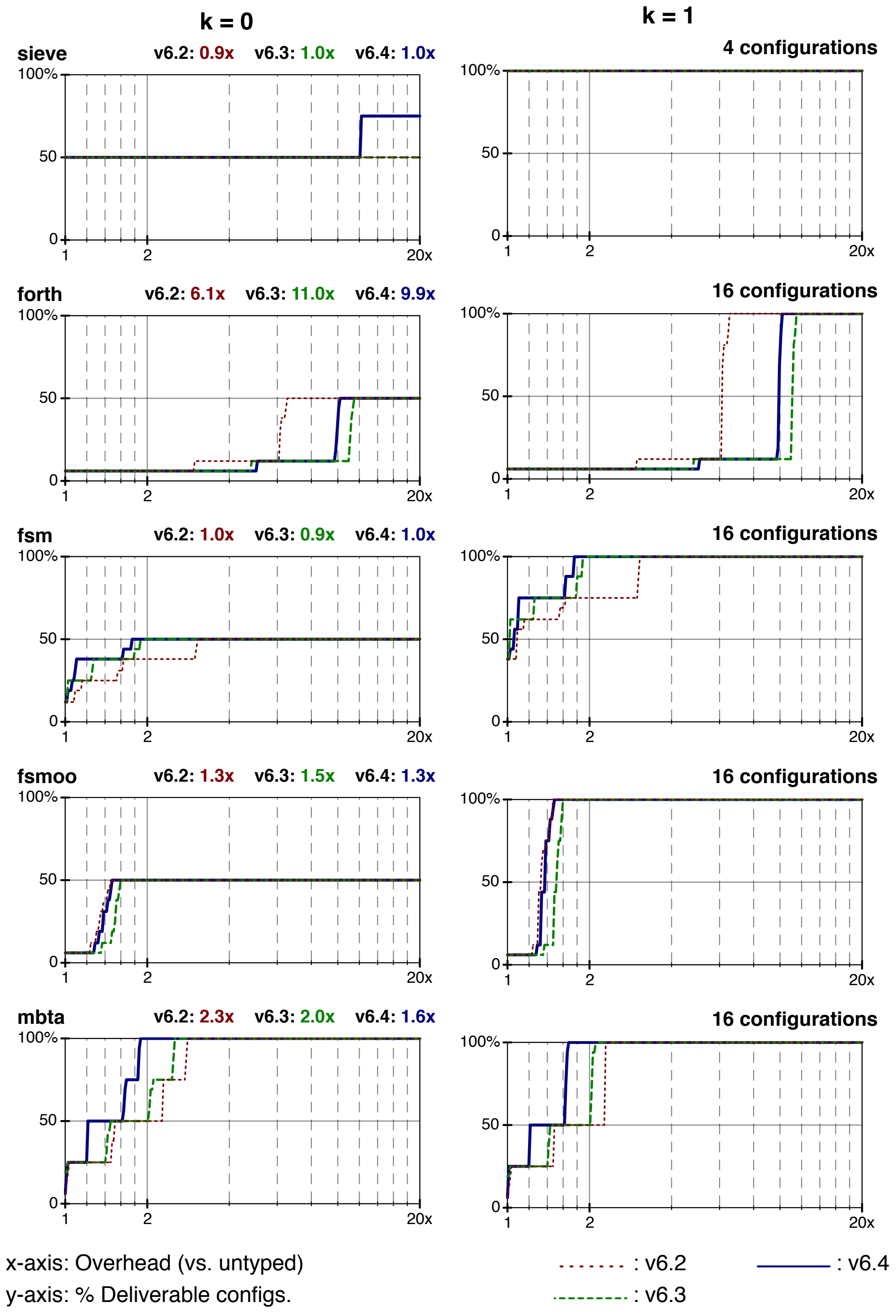


Figure 6: GTP overhead plots (1/4)

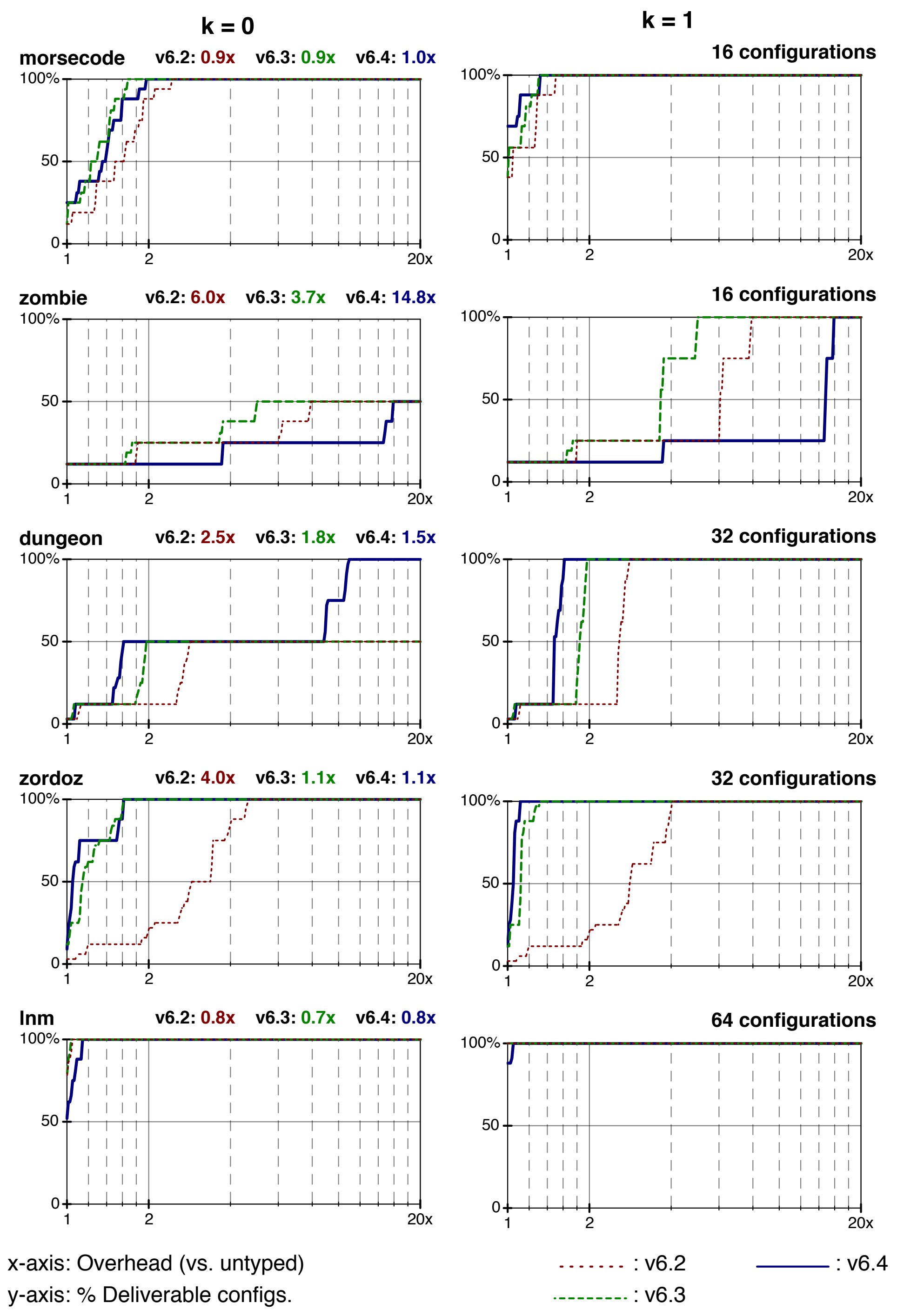


Figure 7: GTP overhead plots (2/4)

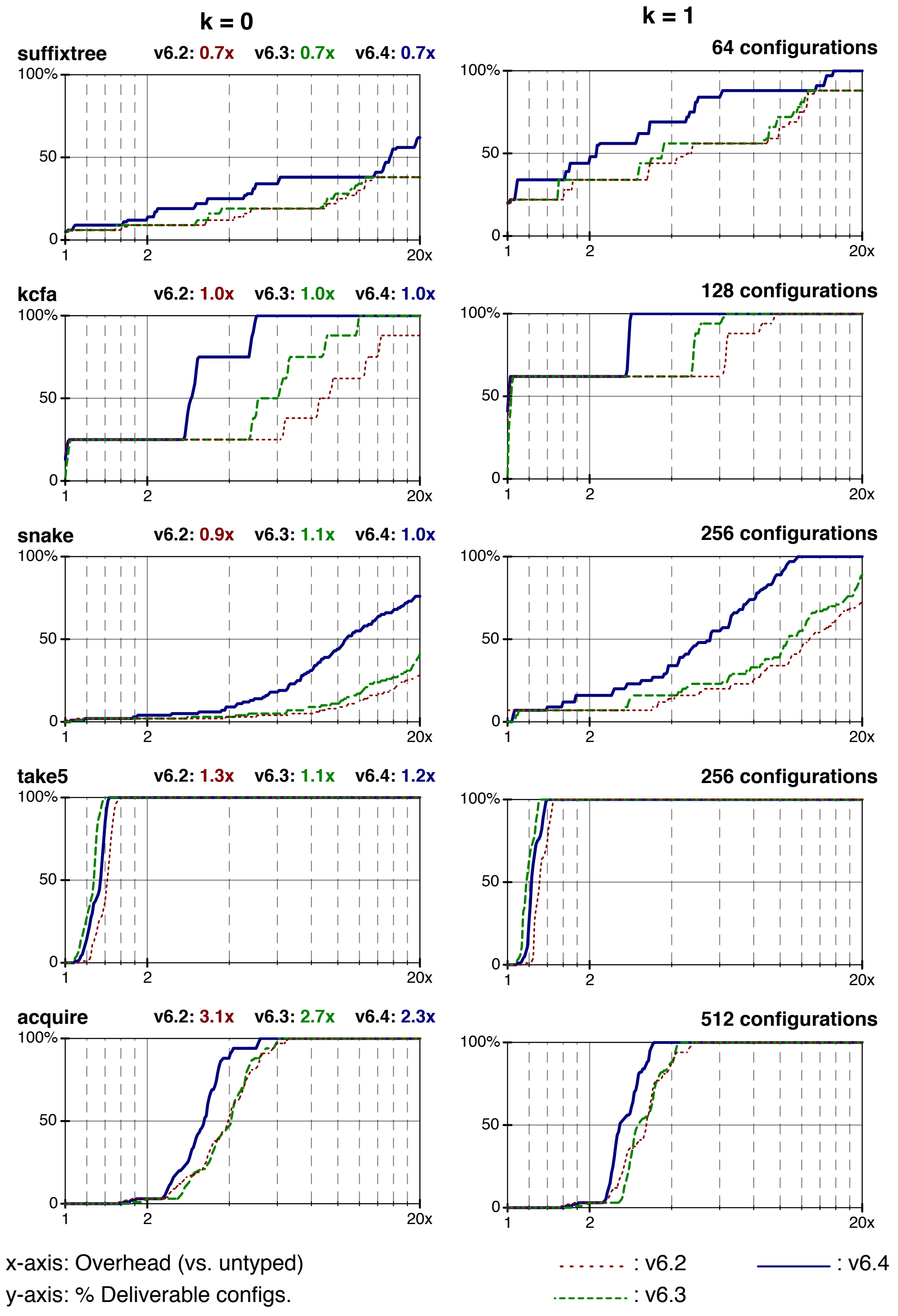


Figure 8: GTP overhead plots (3/4)

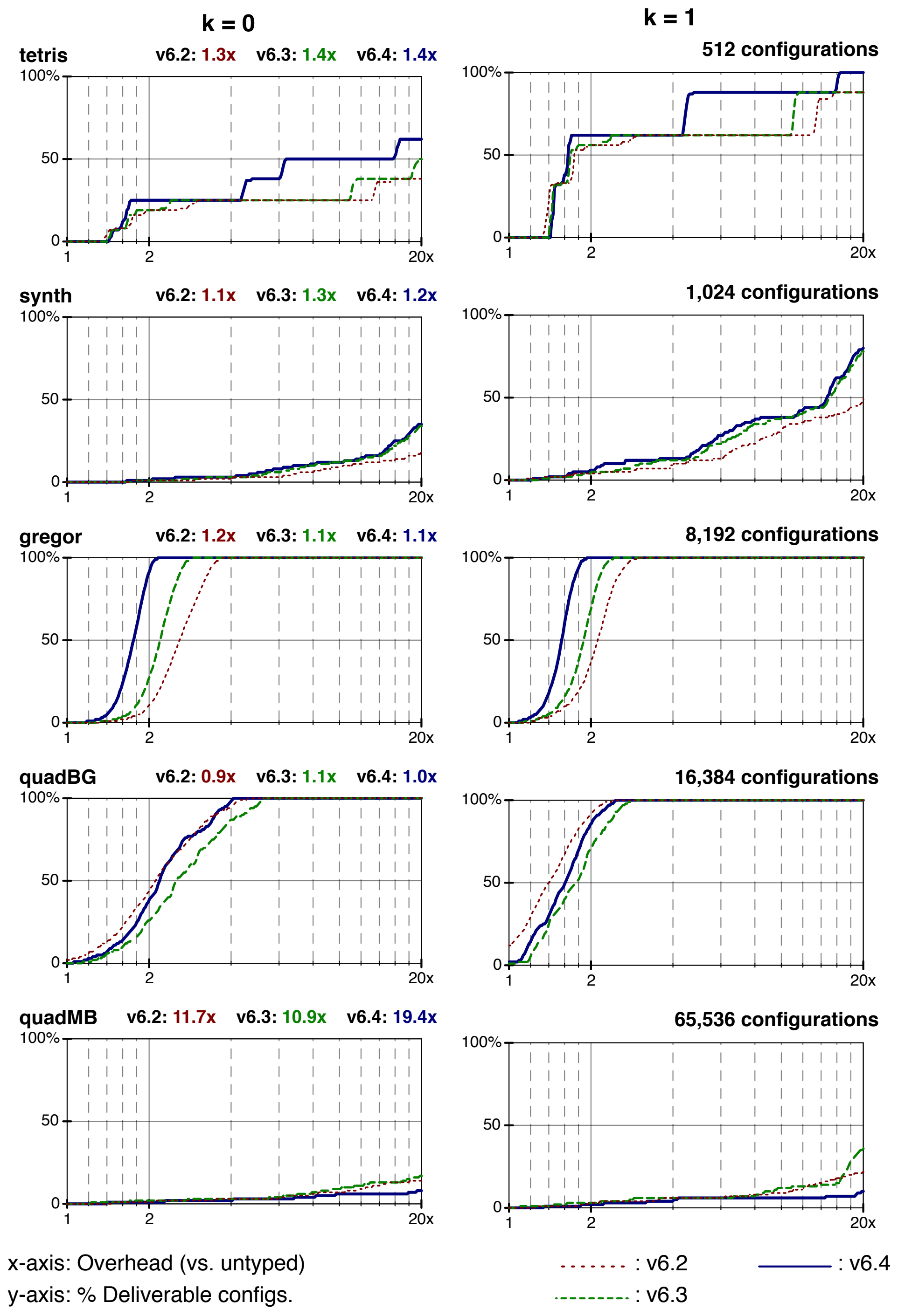


Figure 9: GTP overhead plots (4/4)

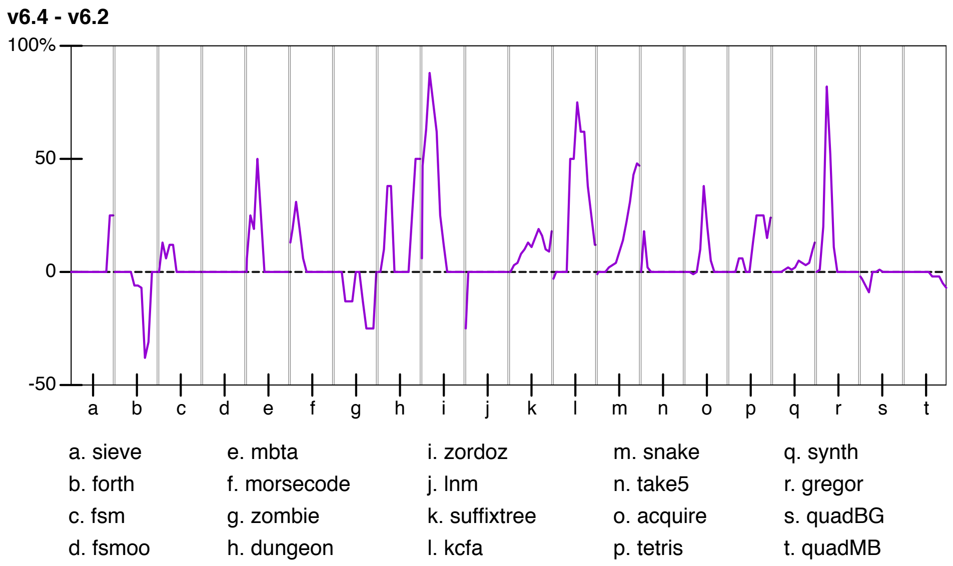


Figure 10: Relative performance of v6.4 versus v6.2

The plot of figure 10 explicitly shows the improvement of version 6.4 over version 6.2. It consists of twenty purple lines, one for each benchmark. These lines plot the difference between the curves for v6.4 and v6.2 on the corresponding overhead plot. For example, the line for `gregor` (labeled r) demonstrates a large improvement in the number of 2-deliverable configurations. The plot also shows that fifteen of the twenty benchmarks significantly benefit from running on version 6.4. Only the line for the `forth` benchmark demonstrates a significant regression; the `zombie` benchmark demonstrates a small regression due to an increase in the cost of type casts.

The improved performance of Racket version 6.4 is due to revisions of the contract system and Typed Racket's use of contracts to enforce static types. In particular, the contract system allocates fewer closures to track the labels that Typed Racket uses to report type boundary errors. The regression in the `forth` benchmark is due to a bug in the implementation of class contracts in version 6.2. This bug would suppress the allocation of certain necessary class contracts. With the bug fixed, `forth` generates the contracts but suffers additional performance overhead.

6 Evaluation Method, Part II

The evaluation method of section 3 does not scale to benchmarks with a large number of migratable modules. Benchmarking a full performance lattice for a program with N such components requires 2^N measurements. In practice, this limits an exhaustive evaluation of Typed Racket to programs with approximately 20 migratable modules. An evaluation of micro-level gradual typing would be severly limited; depending on the definition of a migratable component, such an evaluation might be limited to programs with 20 functions.

ZU064-05-FPR paper 27 September 2018 21:12

Fortunately, simple random sampling can approximate the ground truth presented in section 5. Instead of measuring every configuration in a benchmark, it suffices to randomly sample a linear number of configurations and plot the overhead apparent in the sample.

Figure 11 plots the true performance of the `snake` benchmark against confidence intervals (Neyman 1937) generated from random samples. The plot on the left shows the absolute performance of `snake` on version 6.2 (dashed red line) and version 6.4 (solid blue line). The plot on the right shows the improvement of version 6.4 relative to version 6.2 (solid purple line). Each line is surrounded by a thin interval generated from five samples of 80 configurations each.

The plots in figure 11 suggest that the intervals provide a reasonable approximation of the performance of the `snake` benchmark. These intervals capture both the absolute performance (left plot) and relative performance (right plot) of `snake`.

Figure 12 provides evidence for the linear sampling suggestion of figure 11 using data for the eleven largest benchmarks in the GTP suite. The solid purple lines from figure 10 alongside confidence intervals generated from a small number of samples. Specifically, the interval for a benchmark with N modules is generated from five samples of $10N$ configurations. Hence the samples for `lnm` use 60 configurations and the samples for `quadMB` use 160 configurations. For every benchmark, the true relative performance (solid purple line) lies within the corresponding interval. In conclusion, a language designer can quickly approximate performance by computing a similar interval.

6.1 Statistical Protocol

For readers interested in reproducing the above results, this section describes the protocol that generated figure 11. The details for figure 12 are analogous:

- To generate one random sample, select 80 configurations (10 times the number of modules) without replacement and associate each configuration with its overhead from the exhaustive performance evaluation reported in section 5.
- To generate a confidence interval for the number of D-deliverable configurations based on five such samples, calculate the proportion of D-deliverable configurations in each sample and generate a 95% confidence interval from the proportions. This is the so-called *index method* (Franz 2007) for computing a confidence interval from a sequence of ratios. This method is intuitive, but admittedly less precise than a method such as Fieller's (Fieller 1957). The two intervals in the left half of figure 11 are a sequence of such confidence intervals.
- To generate an interval for the difference between the number of D-deliverable configurations on version 6.4 and the number of D-deliverable configurations on version 6.2, compute two confidence intervals as described in the previous step and plot the largest and smallest difference between these intervals.

  In terms of figure 12 the upper bound for the number of D-deliverable configurations on the right half of figure 11 is the difference between the upper confidence limit on the number of D-deliverable configurations in version 6.4 and the lower confidence limit on the number of D-deliverable configurations in version 6.2. The corresponding lower bound is the difference between the lower confidence limit on version 6.4 and the upper confidence limit on version 6.2.

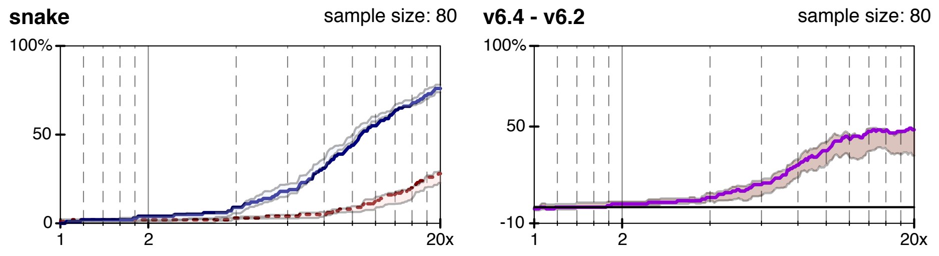


Figure 11: Approximating absolute performance

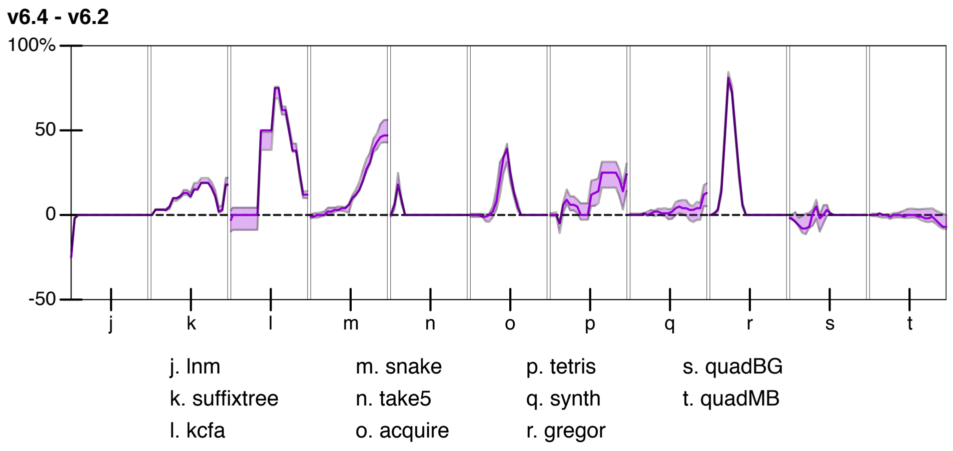


Figure 12: Approximating relative performance

7 Threats to Validity

Although the evaluation method addresses the goals of gradual typing, its application in section 5 manifests a few threats to validity. In particular, both the *experimental protocol* and the *conclusions* have limitations.

There are three significant threats to the protocol. First, some benchmarks have minor interactions with the filesystem. The following benchmarks read from a configuration file: `forth`, `zordoz`, `lnm`, `suffixtree`, `snake`, and `tetris`. The following benchmarks write output to a file: `sieve`, `quadBG`, and `quadMB`. Removing these I/O actions does not change the overhead presented in section 5.2, thus we consider our results representative.

Second, the `quadBG` and `quadMB` configurations that ran in parallel referenced the same Racket executable and external libraries. This cross-reference is a potential source of bias, but we have been unable to detect adverse effects.

Third, the protocol of compiling and running *once* before collecting data is a simplistic way to control for the effects of JIT compilation. Nevertheless, the overheads evident in the results are much larger than those attributed to systematic biases in the literature (Curtsinger and Berger 2013; Gu et al. 2005; Mytkowicz et al. 2009).

The conclusions have five limitations. First, the evaluation does not systematically measure the effects of annotating the same code with different types. This is an issue because type annotations determine the runtime constraints on untyped code. Therefore if two programmers give the same code different type annotations, they may experience different performance problems. For example, `quadBG` and `quadMB` describe the same code with different types and have extremely different performance characteristics. Whereas all configurations of the former are 6-deliverable, only a small fraction of `quadMB` configurations are 20-deliverable.

Second, the benchmark programs are relatively small. Larger programs might avoid the pathological overheads in the benchmarks, though the results for `quadMB` and `synth` constitute evidence to the contrary.

Third, the evaluation does not vary the inputs to the benchmark programs; the conclusions are therefore based on one trace through each program. We consider these traces representative, but some users may observe different traces that stress a different set of type boundaries.

Fourth, the conclusions rely on Typed Racket's implementation technology and do not necessarily generalize to other implementations of gradual typing. Typed Racket re-uses Racket's runtime, a conventional JIT technology. In particular, the JIT makes no attempt to reduce the overhead of contracts. Contract-aware implementation techniques such soft contracts (Nguyễn et al. 2014) or the Pycket tracing JIT compiler (Bauman et al. 2015; Bauman et al. 2017) may significantly reduce the overhead of gradual typing.

Finally, Typed Racket relies on the *complete monitoring* property (Dimoulas et al. 2012) of the Racket contract system to provide a strong form of type soundness (Tobin-Hochstadt et al. 2017). When a type boundary error occurs, Racket produces the original type annotation and the dynamically-typed value that violates the annotation. This protocol generates extremely precise error messages, but the runtime system must dynamically track contextual information to implement it. On one hand, there may be inefficiencies in Racket's implementation of this runtime monitoring. On the other hand, a different gradual type system could offer a different soundness guarantee and circumvent the need for this runtime accounting altogether. For example, Thorn (Bloom et al. 2009) and StrongScript (Richards et al. 2015) let the programmer chose whether to enforce a type annotation at runtime. Reticulated Python provides an alternative, type-tag notion of soundness (Vitousek et al. 2014; Vitousek et al. 2017) that may impose less overhead (Greenman and Migeed 2018), but gives programmers far less precise information than Typed Racket for debugging type boundary errors.

8 Dissecting Performance Overhead

Our evaluation demonstrates that adding types to an arbitrarily chosen subset of Racket modules in a program can severely degrade the performance of the program. This section explains the aspects of Typed Racket that influence performance and investigates sources of high overhead in the benchmark programs.

8.1 How Typed Racket Enforces Type Soundness

Performance overhead in Typed Racket is due to its type soundness guarantee. When a value flows from Racket to a typed context, there is a runtime cost to check that the value matches the assumptions of the static type checker.

Typed Racket's strategy for checking the type of a Racket value at runtime follows the *natural embedding* technique for a multi-language program (Matthews and Findler 2009). When a typed context expects a value of base type, Typed Racket enforces the boundary with a predicate for the type. When a typed context expects a value of an inductive type, Typed Racket checks the constructor of the value and recursively checks its components. Lastly, when a typed context expects a value of a coinductive type (e.g. a mutable cell or a function), Typed Racket checks the constructor, recursively checks its components when possible, and wraps the incoming value with a proxy (Strickland et al. 2012) to monitor its future interactions with typed code.

To illustrate this strategy, we describe for a few types τ how Typed Racket enforces the invariant by applying a contract $(\!|\tau|\!)$ when an untyped value v flows into a typed region:[13]

- If τ is $\mathsf{Integer}$ then $((\!|\tau|\!)\,v)$ checks that v is an integer constant.
- If τ is $\mathsf{Boolean} \cup \mathsf{Integer}$ then $((\!|\tau|\!)\,v)$ checks that v is either a boolean constant or an integer constant.
- If τ is $(\mathsf{Listof\,Integer})$ then $((\!|\tau|\!)\,v)$ checks that v is a list and checks $(\!|\mathsf{Integer}|\!)$ for every value in the list.
- If τ is $(\mathsf{Vectorof\,Integer})$ then $((\!|\tau|\!)\,v)$ checks that v is a vector, checks $(\!|\mathsf{Integer}|\!)$ for every value in the vector, and wraps v in a proxy that checks $(\!|\mathsf{Integer}|\!)$ for every read from the vector.
- If τ is $(\mathsf{Integer} \rightarrow \mathsf{Integer})$ then $((\!|\tau|\!)\,v)$ checks that v is a function and wraps v in a proxy that checks $(\!|\mathsf{Integer}|\!)$ for every result computed by the function.

Note that the cost of checking a type like $\tau_0 \cup \tau_1$ is linear in the number of types in the union, and the cost of a type like $(\mathsf{Listof}\,\tau)$ is linear in the size of the list value. Furthermore, if $((\!|\tau|\!)\,v)$ wraps v in a proxy then $((\!|\tau|\!)\,((\!|\tau|\!)\,v))$ may wrap v in two proxies, and thereby add two levels of indirection. See Tobin-Hochstadt and Felleisen (2006) and Takikawa et al. (2015) for additional details regarding the type-to-contract translation.

This strategy clearly suffers from three kinds of performance costs: the cost of checking a value as it crosses a boundary, the cost of allocating a proxy to monitor the value, and the cost of the indirection that the proxy adds to subsequent operations. The rest of this section demonstrates how these costs arise in practical examples.

8.2 High-Frequency Typechecking

If values frequently cross one type boundary, then the program suffers even if the cost of each boundary-crossing is relatively low. The program in figure 13, for example, calls the typed function `stack-empty?` one million times from untyped code. Each call is type-correct; nevertheless, Typed Racket validates the argument `stk` against the specification $(\!|$`Listof A`$|\!)$ one million times.

[13] When a typed value flows into an untyped region, Typed Racket dualizes these contracts.

```
#lang typed/racket
(provide:
  [stack-empty?
   (-> Stack Boolean)])

(define-type Stack
  (Listof Integer))

(define (stack-empty? stk)
  (null? stk))
```

```
#lang racket
(require 'stack)

;; Create a stack of 20 elements
(define stk (range 20))

(for ([i (in-range 10e6)])
  (stack-empty? stk))
```

Figure 13: A high-frequency type boundary

High-frequency module boundaries are common in the GTP benchmarks. To give an extreme example, over six million values flow across four separate boundaries in `snake`. In `suffixtree`, over one hundred million values flow across two boundaries. When these module boundaries are type boundaries, the benchmarks suffer considerable overhead; their respective worst cases are 32x and 28x on Racket v6.4.

8.3 High-Cost Types

Certain types require computationally expensive runtime checks in Typed Racket. Immutable lists require a linear number of checks. Functions require proxies, whose total cost then depends on the number of subsequent calls. Mutable data structures (hash tables, objects) are the worst of both worlds, as they require a linear number of such proxies.

In general Typed Racket programmers are aware of these costs, but predicting the cost of enforcing a specific type in a specific program is difficult. One example comes from `quadMB`, in which the core datatype is a tagged *n*-ary tree type. Heavy use of a predicate for this type causes the 19.4x typed/untyped ratio in Racket v6.4. Another example is the `kcfa` benchmark, in which hashtable types account for up to a 3x slowdown.

High-cost types may also come from a library that the programmer has no control over. For example, the script in figure 14 executes in approximately twelve seconds. Changing its language to `#lang typed/racket` improves performance to under 1 millisecond by removing a type boundary to the `trie` library.[14] Michael Ballantyne encountered a similar issue with a queue library that led to a 1275x slowdown (see Appendix).

8.4 Complex Type Boundaries

Higher-order values and metaprogramming features introduce fine-grained, dynamic type boundaries. For example, every proxy that enforces a type specification is a dynamically-generated type boundary. These boundaries make it difficult to statically predict the overhead of gradual typing.

[14] There is no way for a programmer to predict that the dynamic check for the `trie` type is expensive, short of reading the implementation of Typed Racket and the `pfds/trie` library.

```
#lang racket
(require pfds/trie) ;; a Typed Racket library

(define t (trie (list (range 128))))
(time (bind (range 128) 0 t))
```

Figure 14: Performance pitfall, discovered by John Clements.

```
#lang typed/racket
(require (prefix-in P. "population.rkt"))

(: evolve (P.Population Natural -> Real))
(define (evolve p iters)
  (cond
    [(zero? iters) (get-payoff p)]
    [else (define p2 (P.match-up* p r))
          (define p3 (P.death-birth p2 s))
          (evolve p3 (- iters 1))]))
```

Figure 15: Accumulating proxies in `fsm`.

The `synth` benchmark illustrates one problematic use of metaprogramming. One module in `synth` exports a macro that expands to a low-level iteration construct. The expanded code introduces a reference to a server module, whether or not the macro client statically imports the server. Thus, when the server and client are separated by a type boundary, the macro inserts a type boundary in the expanded looping code. In order to predict such costs, a programmer must recognize macros and understand each macro's namespace.

```
#lang typed/racket
(require (prefix-in C. "command.rkt"))

(: eval (Input-Port -> Env))
(define (eval input)
  (for/fold ([env  : C.Env  (base-env)])
            ([line : String (in-lines input)])
    ;; Cycle through commands in `env` until
    ;;  `eval-line` gives a non-#f result
    (for/first ([c : (Instance C.Cmd%) (in-list env)])
      (send c eval-line env line))))
```

Figure 16: Accumulating proxies in `forth`.

8.5 Layered Proxies

Higher-order values that repeatedly cross type boundaries may accumulate layers of type-checking proxies. These proxies add indirection and space overhead.

```
#lang typed/racket

(define-type Posn ((U 'x 'y 'move) ->
                   (U (List 'x (-> Natural))
                      (List 'y (-> Natural))
                      (List 'move (-> Natural Natural Posn)))))

(: posn-move (Posn Natural Natural -> Posn))
(define (posn-move p x y)
  (define key 'move)
  (define r (p key))
  (if (eq? (first r) key)
    ((second r) x y)
    (error 'key-error)))
```

Figure 17: Adapted from the `zombie` benchmark.

Racket's proxies implement a predicate that tells whether the current proxy subsumes another proxy. These predicates often remove unnecessary indirections, but a few of the benchmarks still suffer from redundant layers of proxies.

For example, the `fsm`, `fsmoo`, and `forth` benchmarks update mutable data structures in a loop. Figures 15 and 16 demonstrate the problematic functions in each benchmark. In `fsm`, the value `p` accumulates one proxy every time it crosses a type boundary; that is, four proxies for each iteration of `evolve`. The worst case overhead for this benchmark is 235x on Racket v6.4. In `forth`, the loop functionally updates an environment `env` of calculator command objects; its worst-case overhead is 27x on Racket v6.4.[15]

The `zombie` benchmark exhibits similar overhead due to higher-order functions. For example, the `Posn` datatype in figure 17 is a higher-order function that responds to symbols `'x`, `'y`, and `'move` with a tagged method. Helper functions such as `posn-move` manage tags on behalf of clients, but calling such functions across a type boundary leads to layered proxies. This benchmark replays a sequence of a mere 100 commands yet reports a worst-case overhead of 300x on Racket v6.4.

8.6 Library Boundaries

Racket libraries are either typed or untyped; there is no middle ground, therefore one class of library clients must communicate across a type boundary. For instance, the `mbta` and `zordoz` benchmarks rely on untyped libraries and consequently have relatively high typed/untyped ratios on Racket v6.2 (2.28x and 4.01x, respectively). In contrast, the `lnm` benchmark relies on two typed libraries and runs significantly faster when fully typed.

[15] Modifying both functions to use an imperative message-passing style removes the performance overhead, though it is a failure of gradual typing if programmers must resort to such refactorings.

9 The Future of Gradual Typing

Gradual typing emerged as a new research area ten years ago. Researchers began by asking whether one could design a programming language with a sound type system that integrated untyped components (Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006). The initial series of papers on gradual typing provided theoretical models that served as proofs of concept. Eight years ago, Tobin-Hochstadt and Felleisen (2008) introduced Typed Racket, the first implementation of a gradual type system designed to accomodate existing, dynamically-typed programs. At this point, the important research question changed to whether the models of gradual typing could scale to express the *grown* idioms found in dynamic languages. Others have since explored this and similar questions in languages ranging from Smalltalk to JavaScript (Allende et al. 2013; Rastogi et al. 2015; Richards et al. 2015; Vitousek et al. 2014).

From the beginning, researchers speculated that the cost of enforcing type soundness at runtime would be high. Tobin-Hochstadt and Felleisen (2006) anticipated this cost and attempted to reduce it by permitting only module-level type boundaries. Herman et al. (2010) and Siek and Wadler (2010) developed calculi to remove the space-inefficiency apparent in models of gradual typing. Industry labs instead built languages with *optional* type annotations (Moon 1974; Steele 1990) that provide static checks but sacrifice type soundness (see TypeScript, Hack, Flow, and mypy). Programs written in such languages run with zero overhead, but are suceptible to silent failures (Tobin-Hochstadt et al. 2017). At least three research languages attempted to provide a middle ground (Bloom et al. 2009; Richards et al. 2015; Vitousek et al. 2017).

As implementations of gradual typing matured, programmers using them had mixed experiences about the performance overhead of gradual typing. Some programmers did not notice any significant overhead. Others experienced orders-of-magnitude slowdowns. The burning question thus became *how to systematically measure* the performance overhead of a gradual type system. This paper provides an answer:

1. To *measure* the performance of a gradual type system, fully annotate a suite of representative benchmark programs and measure the running time of all typed/untyped configurations according to a fixed *granularity*. In Typed Racket, the granularity is by-module. In a micro-level gradual type system such as Reticulated Python, experimenters may choose by-module, by-variable, or any granularity in between. If an exhaustive evaluation is not feasible, the sampling technique of section 6 provides an approximate measure.
2. To express the *absolute performance* of the gradual type system, report the proportion of configurations in each benchmark that are k-step D-deliverable using *overhead plots*. Ideally, many configurations should be 0-step 1.1-deliverable.
3. To express the *relative performance* of two implementations of a gradual type system, plot two overhead plots on the same axis and test whether the distance is statistically significant. Ideally, the curve for the improved system should demonstrate a complete and significant “left shift.”

Applying the evaluation method to Typed Racket has confirmed that the method works well to uncover performance issues in a gradual type system and to quantify improvements between distinct implementations of the same gradual type system.

The results of the evaluation in section 5 suggest three vectors of future research for gradual typing in general. The first vector is to evaluate other gradual type systems. The second is to apply the sampling technique to large applications and to micro-level gradual typing. The third is to build tools that help developers navigate a performance lattice, such as the feature-specific profiler of St-Amour et al. (2015). Greenman and Migeed (2018) have made progress on the first and second vectors with a performance evaluation of Reticulated; consequently, Greenman and Felleisen (2018) have confirmed that porting Reticulated's notion of soundness (Vitousek et al. 2017) to Typed Racket can improve its performance at the cost of detailed error messages. These are encouraging first steps.

Typed Racket in particular must address the pathologies identified in section 8. Here are a few suggestions. To reduce the cost of high-frequency checks, the runtime system could cache the results of successful checks (Ren and Foster 2016) or implement a tracing JIT compiler tailored to identify dynamic type assertions (Bauman et al. 2015; Bauman et al. 2017). High-cost types may be a symptom of inefficiencies in the translation from types to dynamic checks. Recent calculi for space-efficient contracts (Garcia 2013; Greenberg 2015, 2016; Herman et al. 2010; Siek et al. 2015a) may provide insight for eliminating proxies. Storing runtime type information in the heap may prove to be more efficient than encoding it with contracts (Rastogi et al. 2015; Siek et al. 2015b). Lastly, there is a long history of related work on improving the performance of dynamically typed languages (Consel 1988; Gallesio and Serrano 1995; Henglein 1992; Jagannathan and Wright 1995); some of it may apply here.

Finally, researchers must ask whether the specific problems reported in this paper indicate a fundamental limitation of gradual typing. The only way to know is through further systematic evaluation of gradual type systems.

Acknowledgments

The authors gratefully acknowledge support from the National Science Foundation (SHF 1518844) and thank the anonymous JFP reviewers. They owe the implementors of Typed Racket a large debt, especially Sam Tobin-Hochstadt. They also thank Matthew Butterick, John Clements, Matt Might, Linh Chi Nguyen,Vincent St-Amour, Neil Toronto, David Van Horn, Danny Yoo, and Jon Zeppieri for providing programs that inspired the benchmarks. Sam Tobin-Hochstadt and Brian LaChance provided feedback on early drafts.

Bibliography

Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft Typing with Conditional Types. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 163–173, 1994.

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.

Christopher Anderson, Paul Giannini, and Sophia Drossopoulou. Toward Type Inference for JavaScript. In *Proc. European Conference Object-Oriented Programming*, pp. 428–452, 2005.

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.

Spenser Bauman, Sam Tobin-Hochstadt, Jeremy G. Siek, and Carl Friedrich Bolz-Tereick. Sound Gradual Typing: Only Mostly Dead. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2017.

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.

Robert Cartwright and Mike Fagan. Soft Typing. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 278–292, 1991.

Charles Consel. New Insights into Partial Evaluation: the SCHISM Experiment. In *Proc. European Symposium on Programming*, pp. 236–246, 1988.

Charlie Curtsinger and Emery Berger. Stabilizer: Statistically Sound Performance Evaluation. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 219–228, 2013.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Symposium on Programming*, pp. 214–233, 2012.

E.C. Fieller. Some Problems in Interval Estimation. *Journal of the Royal Statistical Society* 16(2), pp. 175–185, 1957.

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 23–32, 1996.

Matthew Flatt. Bindings as Sets of Scopes. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 705–717, 2016.

Volker H. Franz. Ratios: A short guide to confidence limits and proper use. Unpublished Manuscript, 2007.

Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. In *Proc. Symposium on Applied Computing*, pp. 1859–1866, 2009.

Erick Gallesio and Manuel Serrano. Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages. In *Proc. International Static Analysis Symposium*, pp. 366–381, 1995.

Ronald Garcia. Calculating Threesomes, with Blame. In *Proc. ACM International Conference on Functional Programming*, pp. 417–428, 2013.

Ronald Garcia and Matteo Cimini. Principal Type Schemes for Gradual Programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 303–315, 2015.

Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting Gradual Typing. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 429–442, 2016.

Michael Greenberg. Space-Efficient Manifest Contracts. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 181–194, 2015.

Michael Greenberg. Space-Efficient Latent Contracts. In *Proc. Symposium Trends in Functional Programming*, 2016.

Ben Greenman and Matthias Felleisen. The Spectrum of Soundness and Performance. Submitted for review, 2018.

Ben Greenman and Zeina Migeed. On the Cost of Type-Tag Soundness. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2018.

Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code Layout as a Source of Noise in JVM Performance. *Studia Informatica Universalis* 4(1), pp. 83–99, 2005.

Fritz Henglein. Global Tagging Optimization by Type Inference. In *Proc. ACM Symposium on LISP and functional programming*, pp. 205–215, 1992.

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.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-Efficient Gradual Typing. *Higher-Order and Symbolic Programming* 23(2), pp. 167–189, 2010.

Suresh Jagannathan and Andrew K. Wright. Effective Flow Analysis for Avoiding Run-Time Checks. In *Proc. International Static Analysis Symposium*, pp. 207–224, 1995.

Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. Sage: Unified Hybrid Checking for First-Class Types, General Refinement Types, and Dynamic (Extended Report). 2007.

Erwan Lemonnier. Pluto: or how to make Perl juggle with billions. Forum on Free and Open Source Software (FREENIX), 2006. `http://erwan.lemonnier.se/talks/pluto.html`

Jacob Matthews and Robert Bruce Findler. Operational Semantics for Multi-Language Programs. *Transactions on Programming Languages and Systems* 31(3), pp. 12:1–12:44, 2009.

David A. Moon. MACLISP Reference Manual. 1974.

Todd Mytkowicz, Amer Diwan, Matthais Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong. In *Proc. ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 265–276, 2009.

J. Neyman. Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability. *Philosophical Transactions of the Royal Society of London* 236(767), pp. 333–380, 1937.

Linh Chi Nguyen. Tough Behavior in Repeated Bargaining game, A Computer Simulation Study. Master in Economics dissertation, University of Trento, 2014.

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.

Brianna M. Ren and Jeffrey S. Foster. Just-in-time Static Type Checking for Dynamic Languages. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 462–476, 2016.

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

Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and Coercion: Together again for the first time. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 425–435, 2015a.

Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In *Proc. European Symposium on Programming*, pp. 432–456, 2015b.

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

Jeremy G. Siek and Manish Vachharajani. Gradual Typing with Unification-based Inference. In *Proc. Dynamic Languages Symposium*, 2008.

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

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

Guy L. Steele. Common Lisp the Language. 2nd edition. Digital Press, 1990.

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical Variable-Arity Polymorphism. In *Proc. European Symposium on Programming*, pp. 32–46, 2009.

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 943–962, 2012.

Norihisa Suzuki. Inferring Types in Smalltalk. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 187–199, 1981.

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.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 456–468, 2016.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual Typing for First-Class Classes. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 793–810, 2012.

Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining Delimited Control with Contracts. In *Proc. European Symposium on Programming*, pp. 229–248, 2013.

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

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 395–406, 2008.

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten years later. In *Proc. Summit on Advances in Programming Languages*, 2017.

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.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *Proc. ACM Symposium on Principles of Programming Languages*, 2017.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual Typestate. In *Proc. European Conference on Object-Oriented Programming*, pp. 459–483, 2011.

10 Appendix

10.1 Anecdotal Evidence of Performance Costs

The following enumeration presents links to some of the anecdotes that triggered this investigation into the performance of gradual typing:

re: Unsafe version of require/typed?. Neil Toronto. 2015-05-01
`https://groups.google.com/d/msg/racket-users/oo_FQqGVdcI/p4-bqol5hV4J`

re: Unsafe version of require/typed?. Michael Ballantyne. 2015-05-01
`https://groups.google.com/d/msg/racket-users/oo_FQqGVdcI/leUnIAn7yqwJ`

Rocking with Racket. Marc Burns. 2015-09-27
`http://con.racket-lang.org/2015/burns.pdf`

Typed/Untyped cost reduction and experience. John Griffin. 2015-12-26
`https://groups.google.com/d/msg/racket-users/rfM6koVbOS8/klVzjKJ9BgAJ`

warning on use trie functions in #lang racket?. John B. Clements. 2016-01-05
`https://groups.google.com/d/msg/racket-users/WBPCsdae5fs/J7CIOeV-CQAJ`

Generative Art in Racket. Rodrigo Setti. 2016-09-18
`http://con.racket-lang.org/2016/setti.pdf`

The online supplement to this paper includes copies of these email threads and documents.

10.2 The GTP *Benchmarks, by Module*

The following summaries describe the module-level structure of benchmarks in the GTP suite. In particular, the summaries include:

- the name and size of each module;
- information about any adaptor modules;
- the number of identifiers imported and exported by the module;
- and a graph of inter-module dependencies, with edges from each module to the modules it imports from.

The modules in each benchmark are ordered alphabetically. Figure 2 uses this ordering on modules to represent configurations as black and white rectangles. For example, the node in figure 2 in which only the left-most segment is white represents the configuration where module `data.rkt` is untyped and all other modules are typed. Similarly, figure 18 derives a natural number for each configuration using the alphabetical order of module names. Configuration 4 in figure 18 (binary: `0100`) is the configuration where only `main.rkt` is typed.

sieve

0. `main`
1. `streams`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	16	13		9	0
1	19	4		0	9

$$\begin{tikzcd} 0 \arrow[r] & 1 \end{tikzcd}$$

forth

0. `command`
1. `eval`
2. `main`
3. `stack`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	132	14		7	2
1	79	4		9	1
2	9	3		1	0
3	35	9		0	14

$$\begin{tikzcd} 2 \arrow[r] & 1 \arrow[r] \arrow[rr, bend right] & 0 \arrow[r] & 3 \end{tikzcd}$$

fsm

0. `automata`
1. `main`
2. `population`
3. `utilities`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	84	28	✓	0	20
1	24	10		17	0
2	46	12		13	4
3	28	6		0	6

$$\begin{tikzcd}
\textcircled{1} \arrow[r] \arrow[rr, bend left] \arrow[rrd] & \textcircled{2} \arrow[r] \arrow[rd] & \textcircled{0} \\
 & & \textcircled{3}
\end{tikzcd}$$

fsmoo

0. `automata`
1. `main`
2. `population`
3. `utilities`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	111	35	✓	0	5
1	18	11		4	0
2	42	28		8	1
3	23	9		0	6

$$\begin{tikzcd}
\textcircled{1} \arrow[r] \arrow[rrd] & \textcircled{2} \arrow[r] \arrow[rd] & \textcircled{0} \\
 & & \textcircled{3}
\end{tikzcd}$$

mbta

0. `main`
1. `run-t`
2. `t-graph`
3. `t-view`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	41	10		6	0
1	40	4		1	6
2	98	44		0	1
3	87	13		1	1

$$\begin{tikzcd}
\textcircled{0} \arrow[r] & \textcircled{1} \arrow[r] & \textcircled{3} \arrow[r] & \textcircled{2}
\end{tikzcd}$$

morsecode

0. `levenshtein`	2. `morse-code-strings`
1. `main`	3. `morse-code-table`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	88	23		0	13
1	25	6		14	0
2	13	5		1	1
3	33	4		0	1

$$\begin{tikzcd}
\textcircled{1} \arrow[rr] \arrow[dr] & & \textcircled{0} \\
& \textcircled{2} \arrow[r] & \textcircled{3}
\end{tikzcd}$$

zombie

0. `image`	2. `math`
1. `main`	3. `zombie`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	16	4	✓	0	7
1	38	9		3	0
2	12	6		0	5
3	236	8		12	3

$$\begin{tikzcd}
\textcircled{1} \arrow[r] & \textcircled{3} \arrow[r] \arrow[dr] & \textcircled{0} \\
& & \textcircled{2}
\end{tikzcd}$$

dungeon

0. `cell`
1. `grid`
2. `main`
3. `message-queue`
4. `utils`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	114	14		2	38
1	61	12		19	10
2	318	31		35	0
3	9	4		0	2
4	32	7		0	6

$$\begin{tikzcd}
2 \arrow[r] \arrow[rr, bend left] \arrow[drrr] & 1 \arrow[r] & 0 \arrow[r] & 3 \\
 & & & 4
\end{tikzcd}$$

zordoz

0. `main`
1. `zo-find`
2. `zo-shell`
3. `zo-string`
4. `zo-transition`

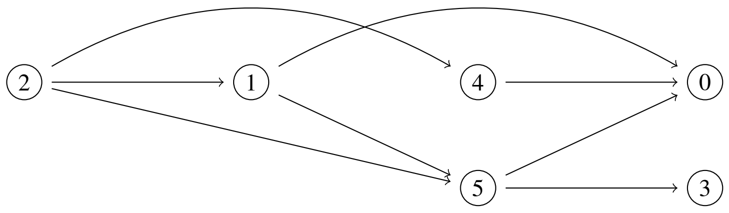
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	15	1		1	0
1	55	16		4	6
2	295	38		10	1
3	613	107		0	6
4	400	53		0	2

$$\begin{tikzcd}
0 \arrow[r] & 2 \arrow[r] \arrow[rr, bend left] \arrow[drr] & 1 \arrow[r] \arrow[dr] & 3 \\
 & & & 4
\end{tikzcd}$$

lnm

0. `bitstring`
1. `lnm-plot`
2. `main`
3. `modulegraph`
4. `spreadsheet`
5. `summary`

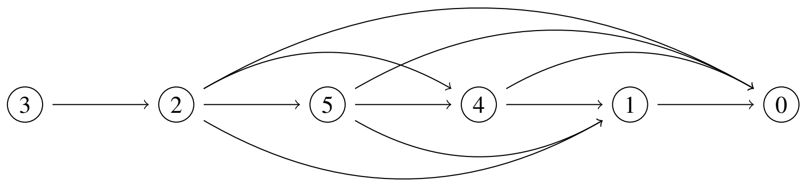
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	36	7		0	12
1	153	41		17	1
2	22	13		15	0
3	142	32	✓	0	9
4	38	8		4	1
5	97	13	✓	13	26



suffixtree

0. `data`
1. `label`
2. `lcs`
3. `main`
4. `structs`
5. `ukkonen`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	8	0	✓	0	108
1	119	40		27	66
2	110	11		66	3
3	13	2		3	0
4	101	40		49	20
5	186	36		59	7



kcfa

0. `ai`
1. `benv`
2. `denotable`
3. `main`
4. `structs`
5. `time`
6. `ui`

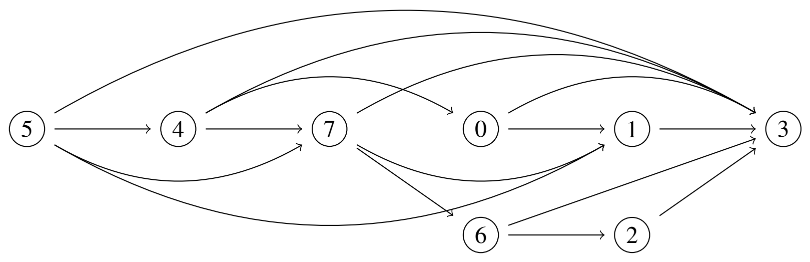
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	49	7		53	3
1	24	7	✓	21	56
2	38	10	✓	39	28
3	29	9		27	0
4	18	0	✓	0	126
5	20	6	✓	35	12
6	51	14		56	6

$$\begin{tikzcd}
3 \arrow[r] \arrow[rrrrrr, bend left=40] & 6 \arrow[r] \arrow[rrrrr, bend left=35] \arrow[rrr, bend left=30] \arrow[rr, bend right=30] \arrow[rrrr, bend right=35] & 0 \arrow[r] \arrow[rrrr, bend left=30] \arrow[rr, bend left=30] \arrow[rrr, bend right=30] & 2 \arrow[r] \arrow[rrr, bend left=30] \arrow[rr, bend right=30] & 5 \arrow[r] \arrow[rr, bend left=30] & 1 \arrow[r] & 4
\end{tikzcd}$$

snake

0. `collide`
1. `const`
2. `cut-tail`
3. `data`
4. `handlers`
5. `main`
6. `motion`
7. `motion-help`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	20	9		21	2
1	17	0		16	15
2	8	1		16	1
3	12	8	✓	0	112
4	16	6		21	2
5	33	10		26	0
6	31	12		23	6
7	23	5		17	2



take5

0. `basics`	4. `deck`
1. `card`	5. `main`
2. `card-pool`	6. `player`
3. `dealer`	7. `stack`

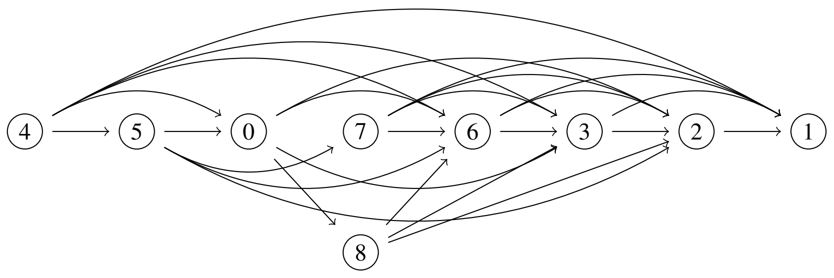
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	25	2		0	24
1	15	2	✓	0	35
2	32	3		15	1
3	101	11		19	1
4	71	4		20	1
5	33	0		3	0
6	27	7		7	4
7	23	-2		7	5

$$\begin{tikzcd}
\circled{5} \arrow[r] \arrow[drrr] & \circled{3} \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left] \arrow[drr] \arrow[drrr] & \circled{4} \arrow[rr, bend left] \arrow[drr] \arrow[ddr] & \circled{1} \arrow[r] \arrow[dr] & \circled{0} \\
 & & & \circled{6} \arrow[r] & \circled{2} \\
 & & & \circled{7} \arrow[ur] &
\end{tikzcd}$$

acquire

0. `admin`	5. `player`
1. `auxiliaries`	6. `state`
2. `basics`	7. `strategy`
3. `board`	8. `tree`
4. `main`	

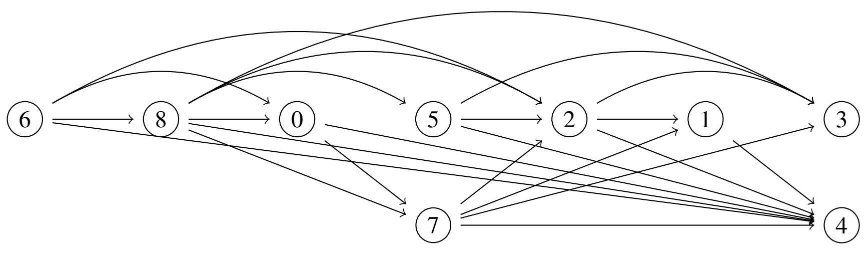
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	262	26		96	4
1	35	4		0	15
2	214	46		3	162
3	433	68	✓	30	150
4	33	7		72	0
5	71	11		65	3
6	359	81	✓	60	170
7	86	16		94	2
8	161	45	✓	91	5



tetris

0. `aux`
1. `block`
2. `bset`
3. `consts`
4. `data`
5. `elim`
6. `main`
7. `tetras`
8. `world`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	22	3		28	6
1	20	7		22	8
2	59	29		29	64
3	7	3		0	12
4	14	6	✓	0	154
5	14	8		41	1
6	21	13		44	0
7	36	10		45	12
8	53	28		51	3



synth

0. `array-broadcast`
1. `array-struct`
2. `array-transform`
3. `array-utils`
4. `data`
5. `drum`
6. `main`
7. `mixer`
8. `sequencer`
9. `synth`

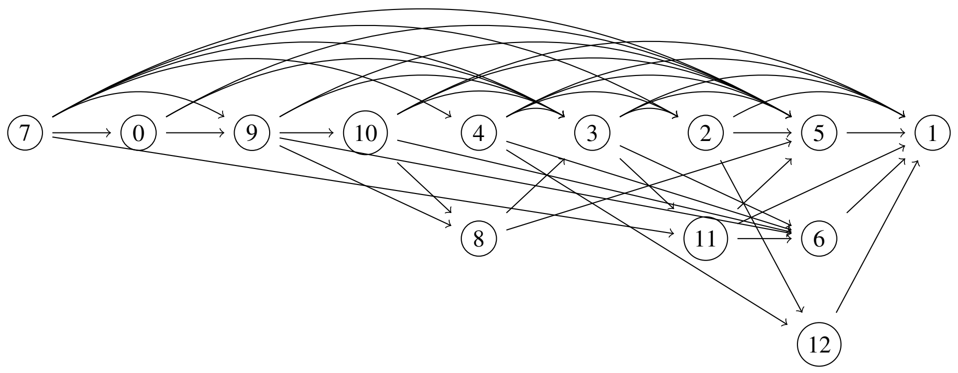
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	102	7		39	6
1	214	9		25	84
2	65	17		42	2
3	132	11		0	45
4	12	0	✓	0	64
5	53	21		44	1
6	94	11		8	0
7	61	17		17	2
8	32	27		20	2
9	70	19		23	12

$$\begin{tikzcd}
6 \arrow[r] \arrow[dr] \arrow[drr] \arrow[drrr] & 5 \arrow[r] \arrow[rrr, bend left] \arrow[rrrr, bend left] \arrow[drr] \arrow[drrrr] & 2 \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left] \arrow[drrr] & 0 \arrow[r] \arrow[rr, bend left] \arrow[drr] & 1 \arrow[r] \arrow[dr] & 3 \\
 & 8 \arrow[ur] \arrow[urrr] \arrow[r] \arrow[rr, bend right] & 7 \arrow[ur] \arrow[urr] & 9 \arrow[ur] \arrow[urr] & & 4
\end{tikzcd}$$

gregor

0. `clock`
1. `core-structs`
2. `date`
3. `datetime`
4. `difference`
5. `gregor-structs`
6. `hmsn`
7. `main`
8. `moment`
9. `moment-base`
10. `offset-resolvers`
11. `time`
12. `ymd`

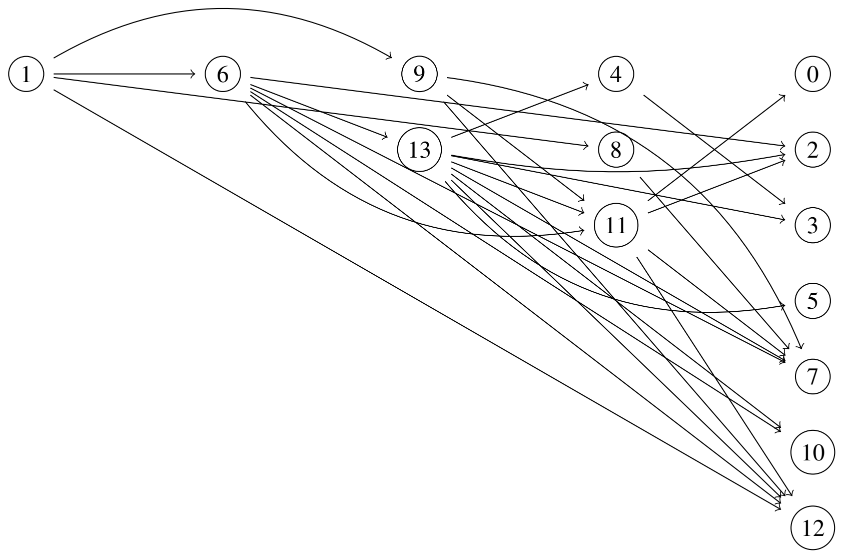
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	57	7		56	16
1	13	-1	✓	0	104
2	70	19		46	30
3	107	24		62	84
4	57	9		79	3
5	18	-1	✓	13	198
6	37	12		13	45
7	127	7		93	0
8	111	32		61	40
9	33	4		36	6
10	90	21		61	13
11	49	14		44	16
12	176	28		13	22



quadBG

0. `hyphenate`
1. `main`
2. `measure`
3. `ocm`
4. `ocm-struct`
5. `penalty-struct`
6. `quad-main`
7. `quads`
8. `quick-sample`
9. `render`
10. `sugar-list`
11. `utils`
12. `world`
13. `wrap`

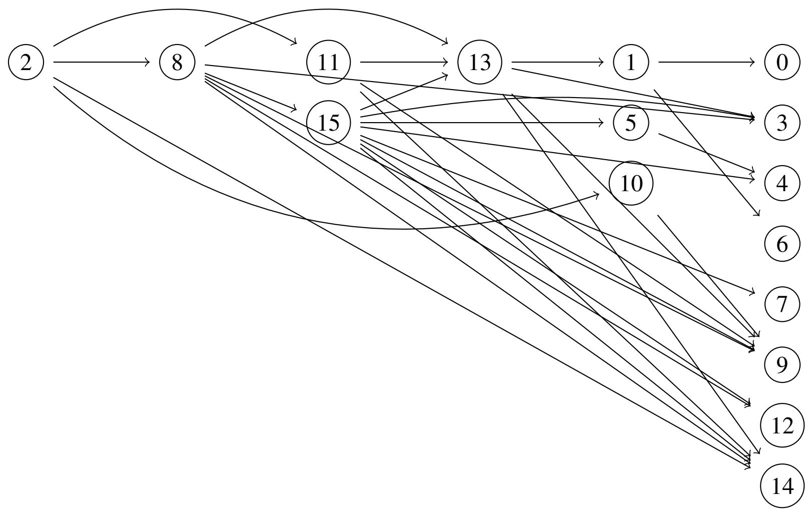
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	5128	42		0	2
1	22	8		71	0
2	56	14		0	15
3	146	10		17	4
4	17	7	✓	0	34
5	6	0	✓	0	5
6	219	25		129	1
7	179	29		0	165
8	30	1		33	1
9	112	2		116	1
10	72	11		0	8
11	212	29		108	45
12	143	1		0	340
13	438	42		151	4



quadMB

0. `exceptions`	8. `quad-main`
1. `hyphenate`	9. `quads`
2. `main`	10. `quick-sample`
3. `measure`	11. `render`
4. `ocm`	12. `sugar-list`
5. `ocm-struct`	13. `utils`
6. `patterns-hashed`	14. `world`
7. `penalty-struct`	15. `wrap`

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	3	1		0	1
1	187	47		2	2
2	20	10		71	0
3	55	13		0	15
4	130	14		17	4
5	17	7	✓	0	34
6	4941	1		0	1
7	5	2	✓	0	5
8	207	39		141	1
9	186	11		0	200
10	31	2		40	1
11	106	10		126	1
12	70	11		0	8
13	179	51		115	54
14	143	30		0	340
15	426	45		161	6



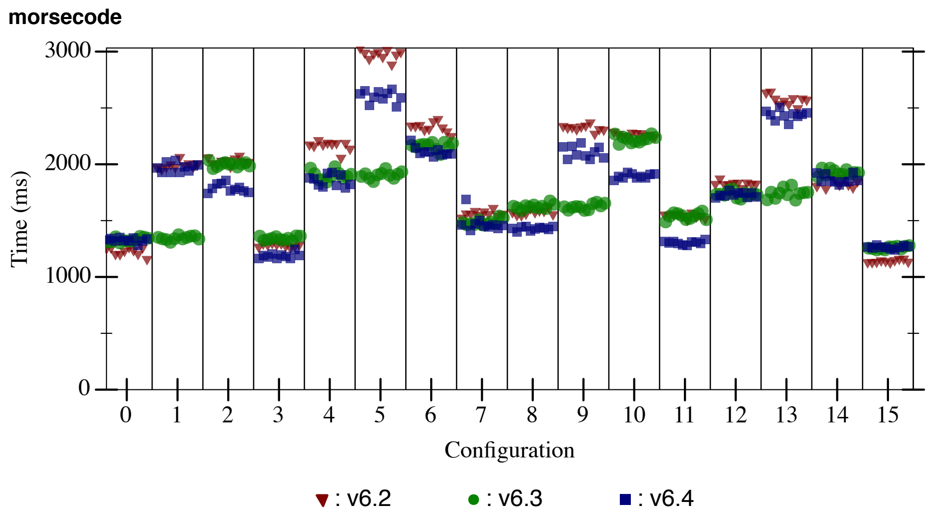


Figure 18: Exact running times in `morsecode`.

10.3 The Stability of Measurements

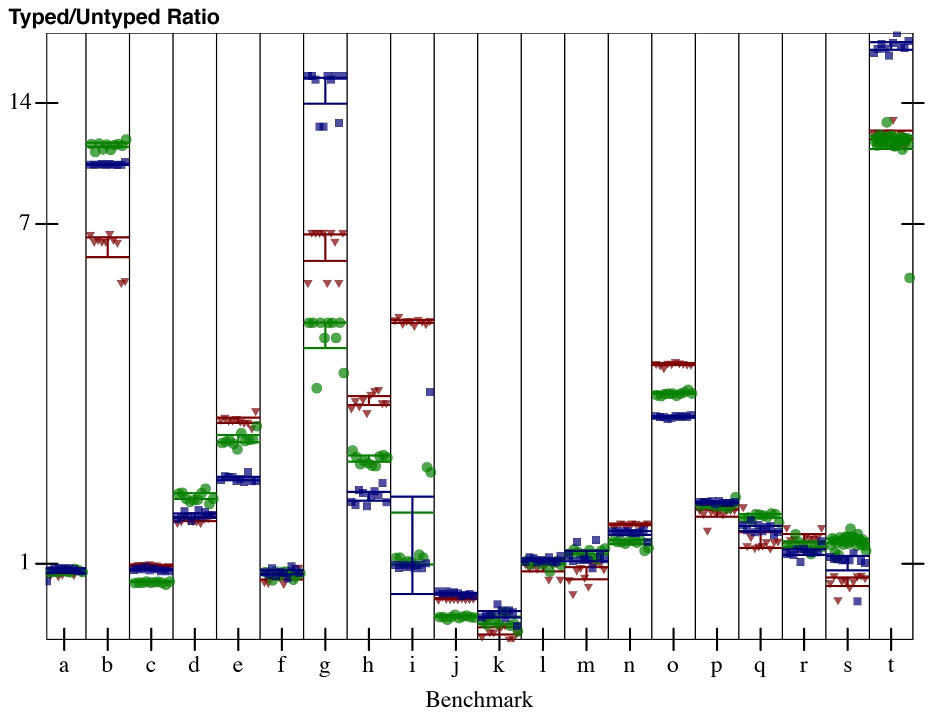
While the experimental setup runs each benchmark multiple times (section 5.1), the overhead plots in section 5.2 use the mean of these running times. The implicit assumption is that the mean of a configuration's running times is an accurate representation of its performance. Figures 18 and 19 qualify this assumption.

Figure 18 plots exact running times for all sixteen `morsecode` configurations. The data for one configuration consists of three sequences of color-coded points; the data for version 6.2 are red triangles, the data for version 6.3 are green circles, and the data for version 6.4 are blue squares. Each sequence of running times is arranged left-to-right in the order in which they were recorded.

For all configurations, the data in each sequence is similar and there is no apparent pattern between the left-to-right order of points and the running time they represent. This suggests that the absolute running times for a given configuration in `morsecode` are independent samples from a population with a stable mean.

Other benchmarks are too large to plot in this manner, but figure 19 plots their exact typed/untyped ratios on a logarithmic scale. Similar to figure 18, the x-axis is segmented; these segments represent the twenty benchmark programs. Within a segment, the color-coded points give the exact typed/untyped ratio from one iteration of the experiment. Finally, each series of points is surrounded by its 95% confidence interval.

Most sequences of points in figure 19 have similar y-values, and none of the sequences evince a strong correlation between their left-to-right (chronological) order and y-value. The notable exception is `quad`. Both `quadBG` and `quadMB` show larger variation between measurements because these measurements were collected on 30 cores running in parallel on the benchmarking machine. The bias is most likely due to contention over shared memory. Nevertheless, figure 19 provides some evidence that the average of a given sequence of typed/untyped ratios is an accurate representation of the true typed/untyped ratio.



▼ : v6.2 ● : v6.3 ■ : v6.4

a. sieve	e. mbta	i. zordoz	m. snake	q. synth
b. forth	f. morsecode	j. lnm	n. take5	r. gregor
c. fsm	g. zombie	k. suffixtree	o. acquire	s. quadBG
d. fsmoo	h. dungeon	l. kcfa	p. tetris	t. quadMB

Figure 19: typed/untyped ratios, on a logarithmic scale.

Benchmark	v6.2	v6.3	v6.4	Benchmark	v6.2	v6.3	v6.4
`sieve`	10	10	10	`suffixtree`	10	10	10
`forth`	10	10	10	`kcfa`	10	10	10
`fsm`	10	10	10	`snake`	10	10	10
`fsmoo`	10	10	11	`take5`	10	10	10
`mbta`	10	10	10	`acquire`	10	10	10
`morsecode`	10	11	10	`tetris`	10	10	10
`zombie`	10	10	10	`synth`	10	10	10
`dungeon`	10	10	10	`gregor`	10	10	11
`zordoz`	10	10	10	`quadBG`	10	30	10
`lnm`	10	10	10	`quadMB`	10	30	10

Figure 20: Samples per benchmark

Benchmark	# Mod.	D = 1	D = 3	D = 5	D = 10	D = 20
`sieve`	2	0	0	0	0	50
`forth`	4	0	0	0	0	0
`fsm`	4	0	0	0	0	0
`fsmoo`	4	0	0	0	0	0
`mbta`	4	0	100	100	100	100
`morsecode`	4	0	100	100	100	100
`zombie`	4	0	0	0	0	0
`dungeon`	5	0	0	0	50	100
`zordoz`	5	0	100	100	100	100
`lnm`	6	17	100	100	100	100
`suffixtree`	6	0	0	0	0	17
`kcfa`	7	0	13	93	100	100
`snake`	8	0	0	0	8	50
`take5`	8	0	100	100	100	100
`acquire`	9	0	2	83	100	100
`tetris`	9	0	0	0	17	17
`synth`	10	0	0	0	0	0

Figure 21: Percent of *D*-deliverable conversion paths, v6.4

10.4 Miscellaneous Figures

The table in figure 20 lists the number of samples per configuration aggregated in section 5.2. For a fixed benchmark and fixed version of Racket, all configurations have an equal number of samples.

The table in figure 21 answers the hypothetical question of whether there exists any "deliverable" conversion paths through a performance lattice in the data for Racket version 6.4. More precisely, a *D*-deliverable *conversion path* for a program of *N* modules is a sequence of *N* configurations $c_1 \rightarrow_1 \ldots \rightarrow_1 c_N$ such that for all *i* between 1 and *N*, configuration c_i is *D*-deliverable. The table lists the number of modules (*N*) rather than the number of paths (*N!*) to save horizontal space.

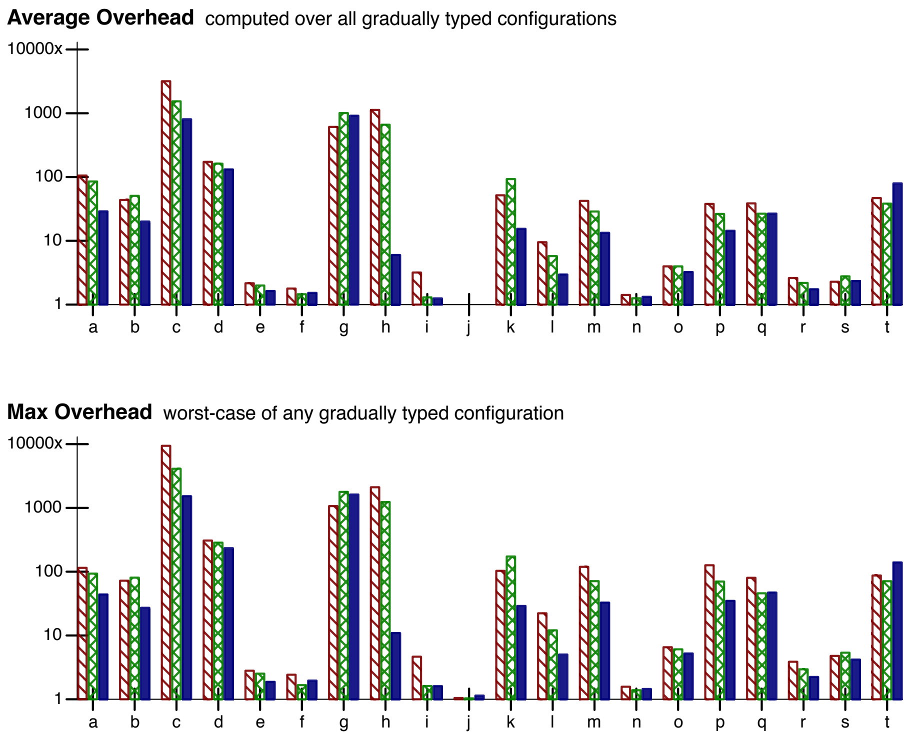
The table in figure 22 provides a few worst-case statistics relevant to section 8. The second column ("Max Boundary") reports the number of times that a value flows across the most-frequently-crossed static module boundary in each benchmark. When this boundary is a type boundary, each of these crossings triggers a runtime check. The remaining columns report properties of the configuration in each benchmark with the highest performance overhead. These properties are:

- The maximum number of function, vector or struct proxies layered on any single value ("Max Wraps"), recorded using the patch described in Strickland et al. (2012).
- The percentage of overall runtime spent checking contracts ("Contract %"), as reported by Racket's contract profiler (St-Amour et al. 2015).
- The percentage of "Contract %" time that was spent checking contracts across a library boundary ("Library %"). Benchmarks that do not interact with third-party libraries have a dash (-) in this column.
- The number of garbage collections recorded during one run ("# GC").

Figure 23 plots the average-case and worst-case overheads in the benchmark programs.

Benchmark	Max Boundary	Max Wraps	Contract %	Library %	#GC
`sieve`	201,692,321	0	59	-	4,705
`forth`	16	3	50	-	15
`fsm`	361,100	4000	25	-	2,329
`fsmoo`	1,100	2	93	-	856
`mbta`	801	2	27	23	38
`morsecode`	821,060	2	18	-	21
`zombie`	13,502	67	78	-	73
`dungeon`	7,392	2	84	-	28
`zordoz`	116,167	2	32	6	33
`lnm`	13,064	2	21	96	27
`suffixtree`	145,826,484	2	95	-	572
`kcfa`	70,670	2	79	-	8,485
`snake`	18,856,600	2	95	-	123
`take5`	39,780	3	66	-	42
`acquire`	13,240	3	40	-	20
`tetris`	82,338,320	2	89	-	212
`synth`	445,637	13	67	-	203
`gregor`	925,521	2	49	5	20
`quadBG`	339,674	390	60	0	69
`quadMB`	281,030	543	81	0	191

Figure 22: Dissecting Performance Overhead



▨ : v6.2
▨ : v6.3
■ : v6.4

a. sieve	e. mbta	i. zordoz	m. snake	q. synth
b. forth	f. morsecode	j. lnm	n. take5	r. gregor
c. fsm	g. zombie	k. suffixtree	o. acquire	s. quadBG
d. fsmoo	h. dungeon	l. kcfa	p. tetris	t. quadMB

Figure 23: Average and worst-case overhead