Under consideration for publication in J. Functional Programming

How to Evaluate the Performance of Gradual Type Systems

BEN GREENMAN[1] ASUMU TAKIKAWA[1] MAX S. NEW[1] DANIEL FELTEY[2]
ROBERT BRUCE FINDLER[2] JAN VITEK[1] and MATTHIAS FELLEISEN[1]
Northeastern University, Boston, Mass.[1] Northwestern University, Chicago, Ill.[2]

Abstract

A sound gradual type system ensures that untyped components of a program can never break the guarantees of statically typed components. Currently, this assurance requires run-time 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 developers who use gradual type systems and the implementors of such systems. Developers cannot predict whether adding types to part of a program will significantly degrade its performance. Implementors cannot precisely determine how improvements to a gradual type system affect the performance of such programs.

This paper presents the first method for evaluating the performance of 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. In order to validate the method, the paper reports on its application to twenty benchmark programs and three versions 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 (Armstrong 2007); Sweden's pension system is a Perl program (Lemonnier 2006), and the server-side applications of some contemporary companies (Dropbox, Facebook, Twitter) are written in dynamic languages (Python, PHP, and Ruby, respectively).

Regardless of why programmers choose dynamically typed languages, the maintainers of these applications inevitably find the lack of explicit type annotations an obstacle to their work. Researchers have tried to overcome the lack of type annotations with inference algorithms (Aiken et al. 1994; Anderson et al. 2005; Cartwright and Fagan 1991; Furr et al. 2009; Henglein and Rehof 1995; Rastogi et al. 2012), but most have come to realize that there is no substitute for programmer-supplied annotations. Explicit annotations communicate a programmer's intent to other human readers. Furthermore, tools can check the annotations for logical inconsistencies and leverage types to improve the efficiency of compiled code.

One solution is to rewrite the entire application in a statically typed language. This solution assumes that the application is small enough and the problem is recognized soon

enough to make a wholesale migration feasible. For example, Twitter was able to port their server-side code from Ruby to Scala because they understood the problem early on.[1] In other cases, the codebase becomes too large for this approach.

Another solution to the problem is gradual typing (Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006),[2] a linguistic approach. In a 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 the soundness of the type annotations.

From a syntactic perspective the interaction is seamless, but dynamic checks introduce runtime overhead. During execution, if an untyped function flows into a variable f with type $\tau_1 \to \tau_2$, then a dynamic check must follow every subsequent call to f because typed code cannot assume that values produced by the untyped function have the syntactic type τ_2. Conversely, typed functions invoked by untyped code must dynamically check their argument values. If functions or large data structures frequently cross these *type boundaries*, enforcing type soundness might impose a huge runtime cost.

Optimistically, researchers have continued to explore the theory and practice of sound gradual typing (Allende et al. 2013; Garcia and Cimini 2015; Knowles et al. 2007; Rastogi et al. 2015; Richards et al. 2015; Siek et al. 2009; Tobin-Hochstadt and Felleisen 2008; Vitousek et al. 2014; Wolff et al. 2011).[3] Some research groups have invested significant resources implementing sound gradual type systems. But surprisingly few groups have evaluated the performance of gradual typing. Most acknowledge an issue with performance in passing. Worse, others report only the performance ratio of fully typed programs relative to fully untyped programs, ironically 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 result with the results of a conference version (Takikawa et al. 2016). Specifically, this paper contributes:

- validation that the method can express the relative performance between three implementations of Typed Racket (section 5.3);
- evidence that simple random sampling can accurately approximate the results of a comprehensive evaluation with asymptotically fewer measurements (section 6);
- eight additional benchmark programs (section 4); and
- a discussion of the pathological overheads in the benchmark programs (section 8).

The discussion is intended to guide implementors of gradual type systems toward promising future directions (section 9).

This paper begins with an extended introduction to our philosophy of gradual typing and the pragmatics of Typed Racket. Section 2.2 in particular argues that type soundness is an imperative, despite the performance cost of enforcing it.

[1] `http://www.artima.com/scalazine/articles/twitter_on_scala.html`
[2] The term *migratory typing* more accurately describes this particular mode of gradual 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

2 Gradual Typing in Typed Racket

Typed Racket (Tobin-Hochstadt and Felleisen 2008) is the oldest and most developed implementation of sound gradual typing. It supports clients in both academia and industry. Typed Racket attracts these clients because it accomodates the idioms of (untyped) Racket; its type system can express concepts such as variable-arity polymorphism (Strickland et al. 2009), first-class classes (Takikawa et al. 2012), and delimited continuations (Takikawa et al. 2013). Finally, a typed module may incorporate definitions from a Racket module with a type-annotated import statement; conversely, a Racket module may use definitions from a Typed Racket module without knowledge that the providing module is typed.

Figure 1 demonstrates gradual typing in Typed Racket with a small application. The untyped 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 checks a given number against this chosen number. The untyped module on the top right implements a naïve player. The driver module at the bottom combines the game and player. It generates a game, prompts `stubborn-player` for ten guesses, and counts the number of correct guesses using the `for/sum` combinator. Unlike the other two, the driver module is implemented in Typed Racket. Typed Racket ensures that the game and player follow the type specifications in the driver's `require/typed` clauses. More precisely, Typed Racket statically checks that the driver module sends only natural numbers to the guessing game and player, and inserts dynamic checks to enforce the return types of `play`, `stubborn-player`, and `check-guess`.

Due to the close integration of Racket and Typed Racket, programmers frequently use both languages within a single application. Furthermore, programmers often migrate Racket

modules to Typed Racket as their application evolves. In general one cannot predict why or how such incremental migrations happen, but here are some common motivations:

- The typechecker provides *assurance* against common bugs.
- Type signatures serve as machine-enforced *documentation*.
- Typed modules gain *performance* improvements from the Typed Racket compiler.
- Adding typed modules reduces *friction* with typed libraries and clients.

Regarding the final point, there are two sources of so-called friction (Barrett et al. 2016) between typed and untyped code. The first is the above-mentioned requirement that typed clients must supply type annotations to use imports from an untyped library. Maintainers of such libraries can instead provide a bridge module with the necessary annotations. The second is the performance overhead of typed/untyped interaction, such as the overhead of dynamically enforcing the return type of `play` in figure 1.

2.1 The Costs of Incremental Typing

Adding types to untyped code is a tradeoff. Performing the type conversion may yield long-term benefits, but incurs three immediate engineering costs.

The first cost is the burden of writing and maintaining type annotations. In particular, Typed Racket is a *macro-level*[4] gradual type system. Every expression within a Typed Racket module must pass the type checker. Consequently, every recursive type in a module needs a declaration and every function parameter, class field, and induction variable needs a type annotation.

The second cost is the risk of introducing bugs during the conversion. Typed Racket mitigates this risk by accomodating Racket idioms, but occasionally programmers must refactor code to satisfy the type checker. Refactoring can always spawn bugs.

The third cost is performance overhead due to typed/untyped interaction. For example, Typed Racket developers have experienced pathologies including a 50% overhead in a commercial web server and 25x-50x slowdowns when using the (typed) math library. Another programmer found that converting a script from Racket to Typed Racket improved its performance from 12 seconds to 1 millisecond.[5]

Of these three costs, the performance overhead is the most troublesome. Part of the issue is that the magnitude of the cost is difficult to predict. Unlike the tangible cost of writing type annotations or the perennial risk of introducing bugs, the performance overhead of enforcing type soundness is not apparent until runtime. Additionally, experience with statically typed languages—or optionally-typed languages such as TypeScript—teaches programmers that type annotations do not impose runtime overhead. Typed Racket insists on gradual type soundness, therefore types need runtime enforcement.

[4] As opposed to *micro-level* gradual typing (Siek et al. 2015).
[5] The appendix contains a list of user reports.

2.2 A Case for Sound Gradual Typing

Recall that types are checkable statements about program expressions. Soundness means that every checked statement holds as the program is executed. Statically typed languages provide this guarantee by type checking every expression. In a gradually typed language, the type system cannot check every expression because some are intentionally untyped. Therefore a sound gradual type system monitors the interaction of typed and untyped program components at runtime. Where typed code claims τ about the value of an untyped expression, the gradual type system inserts a runtime check $(\!|\tau|\!)$ capable of deciding whether an arbitrary value belongs to the denotation of the syntactic type τ. If, at runtime, the expression does not produce a value satisfying $(\!|\tau|\!)$, the program halts with a so-called *type boundary error*.

Remark Type boundary errors arise when type annotations impose new constraints that untyped code does not satisfy. Such errors may indicate latent bugs in the untyped code, but it is equally likely that the new type annotations are incorrect specifications. In other words, the slogan "well-typed programs can't be blamed" (Wadler and Findler 2009) misses the point of gradual typing. ■

Typed Racket implements the runtime check $(\!|\tau|\!)$ for a first-order type τ with a first-order predicate. To enforce a higher-order type, Typed Racket dynamically allocates a proxy to ensure that a value's future interactions with untyped contexts conform with its static type (Findler and Felleisen 2002; Strickland et al. 2012).

Consider the Typed Racket function in figure 2. It implements multiplication for polar-form complex numbers.[6] If an untyped module imports this function, Typed Racket allocates a new proxy. The proxy checks that every value which flows from the untyped module to `polar-*` passes the predicate $(\!|\texttt{C}|\!)$. These checks are indispensible because Typed Racket's static types operate at a higher level of abstraction than Racket's dynamic typing. In particular, the tag checks performed by `first` and `*` do not enforce the `C` type. Without the dynamic checks, the call `(polar-* '(-1 0) '(-3 0))` would produce a well-typed complex number from two ill-typed inputs. Racket's runtime system would not detect this erroneous behavior, therefore the program would silently go wrong. Such are the dangers of committing "moral turpitude" (Reynolds 1983).

Furthermore, even if a dynamic tag check uncovers a logical type error, debugging such errors in a higher-order functional language is often difficult. Well-trained functional programmers follow John Hughes' advice and compose many small, re-usable functions to build a program (Hughes 1989). Unfortunately, this means the root cause of a runtime exception is usually far removed from the point of logical failure.

In contrast, sound gradual typing guarantees that typed code never executes a single instruction using ill-typed values. Programmers can trust that every type annotation is a true statement because the gradual type system inserts runtime checks to remove any doubt. These interposed checks furthermore detect type boundary errors as soon as possible. If such an error occurs, the runtime system points the programmer to the relevant type annotation and supplies the incompatible value as a witness to the logical mistake.

[6] The example is adapted from Reynolds classic paper on types soundness (Reynolds 1983). In practice, Racket and Typed Racket have native support for complex numbers.

```
#lang typed/racket

(provide polar-*)

(define-type C (List Nonnegative-Real Real))
;; C represents complex numbers as
;;   ordered pairs (distance, radians)

(: polar-* (C C -> C))
(define (polar-* c1 c2)
  (list (* (first  c1) (first  c2))
        (+ (second c1) (second c2))))
```

Figure 2: Multiplication for polar form complex numbers

3 Evaluation Method, Part I

Performance evaluation for gradual type systems must reflect how programmers use such systems. Experience with Typed Racket shows that programmers frequently combine typed and untyped code within an application. These applications may undergo incremental transitions that add or remove some type annotations; however, it is rare that a programmer adds explicit annotations to every module in the program all at once. In a typical evolution, programmers compare the performance of the incrementally 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.

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 may convert any subset of the modules in an untyped program to Typed Racket. A comprehensive performance evaluation must therefore consider the space of typed/untyped *configurations* a programmer could possibly create given a full set of type annotations for each module. These configurations form a lattice.

Figure 3 demonstrates one such lattice for a six-module program. 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 in which some (or all) modules are untyped.

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

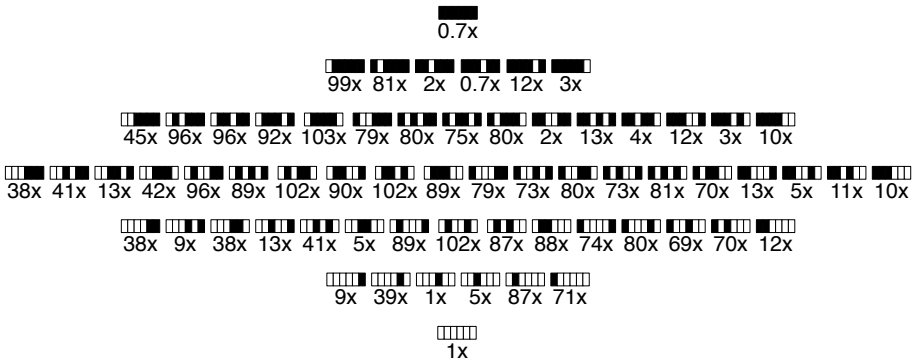


Figure 3: Performance overhead in `suffixtree`, on Racket v6.2.

possible configurations a programmer might encounter after applying three *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 3 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, six 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 (40 configurations have over 20x overhead), but the fully typed configuration runs 30% faster than the untyped configuration because Typed Racket uses the type annotations to compile more efficient code.

Terminology A labeled lattice such as figure 3 is a *performance lattice*. The same lattice without labels is a *configuration lattice*. The practical distinction is that users of a gradual type system will explore configuration lattices and maintainers of such systems may use performance lattices to evaluate overall performance. ■

3.2 Performance Metrics

The most basic question 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 runtime type checks in programs that rely heavily on an untyped library. 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 configuration divided by the time needed to run the bottom 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,

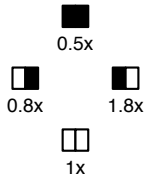


Figure 4: Sample performance lattice

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.

> **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 therefore captures the experience of a *prescient* programmer that converts the k modules suited to improve performance.

Let us illustrate these terms with an example. Suppose there is a project with two modules where the untyped configuration runs in 20 seconds and the typed configuration runs in 10 seconds. Furthermore, suppose the mixed configurations run in 15 and 35 seconds. Figure 4 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. Both mixed configurations are 2-deliverable because they run within 40 seconds, but only one is, e.g., 1.5-deliverable. Lastly, these configurations are 1-step 1-deliverable because they can reach the typed configuration in one type conversion step.

The ratio of *D*-deliverable configurations in such a 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 incrementally 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.

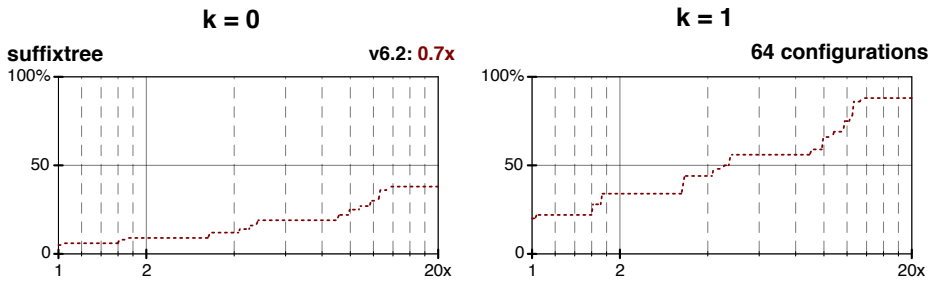


Figure 5: Overhead graphs for `suffixtree`, on Racket v6.2.

3.3 Overhead Graphs

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, 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 number of D-deliverable configurations for various D. The *overhead plot* on the left half of figure 5 presents this information for the performance lattice in figure 3. On the x-axis, possible values for D range continuously from one to twenty. Dashed lines to the left of the 2x tick pinpoint overheads of 1.2x, 1.4x, 1.6x, and 1.8x. To the right of the 2x tick, similar dashed lines pinpoint 4x, 6x, 8x, etc. The y-axis gives the proportion of configurations in the lattice that suffer at most Dx performance overhead. Using this plot, one can confirm the observation made in section 3.1 that six of the 64 configurations (9%) run within a 2x overhead. Additionally, the typed/untyped ratio above the plot reports the 30% performance improvement for the fully typed configuration.

Viewed as a cumulative distribution function, the left plot demonstrates how increasing D increases the number of D-deliverable configurations. In this case, the shallow slope implies that few configurations become deliverable as the programmer accepts a larger performance overhead. The ideal slope would have a steep incline and a large y-intercept, meaning that few configurations have large overhead and many configurations run more efficiently due to the type annotations.

The overhead plot on the right half of figure 5 gives the number of 1-step D-deliverable configurations. A point (X,Y) on this plot represents the percentage Y of configurations c_1 such that there exists a configuration c_2 where $c_1 \rightarrow_1 c_2$ and c_2 runs at most X times slower than the untyped configuration.[7] Intuitively, this plot resembles the (0-step) D-deliverable plot because accounting for one type conversion step does not change the overall characteristics of the benchmark, but only makes more configurations D-deliverable.

These overhead plots concisely summarize the data in figure 3. The same presentation scales to arbitrarily large programs because the y-axis plots the proportion of D-deliverable

[7] Note that c_1 and c_2 may be the same, for instance when c_1 is the fully-typed configuration.

configurations; in contrast, a performance lattice contains exponentially many nodes. Furthermore, plotting the overhead for multiple versions of a gradual type system on the same set of axes provides a high-level summary of the versions' relative performance.

3.3.1 Assumptions and Limitations

Plots in the style of figure 5 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 over 10x overhead. Hence the plots use a log-scaled x-axis to simultaneously encourage fine-grained comparison in the 20-60% 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 3 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 optimistically chooses the best type conversion step. In a program with N modules, a programmer has up to 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 3 but only one of these is 2-deliverable.

4 The GTP Benchmark Programs

The twenty benchmark programs are representative of actual user code yet small enough to make exhaustive performance evaluation tractable. The following descriptions, arranged from smallest performance lattice to largest, briefly summarize each benchmark. 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 : Tool *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 graphs (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

Runs 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 interactions, 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 when arrays frequently cross type boundaries.

gregor

Origin : Library
Purpose : Date and time library
Author : Jon Zeppieri
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 : Library
Purpose : Typesetting
Author : Matthew Butterick
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 does 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 6 tabulates the size and complexity of the benchmark programs.[8] The lines of code (Untyped LOC) and number of 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.[9] 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. Boundaries are import statements from one module to another, excluding imports for 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

[8] The appendix presents the information in figure 6 graphically.
[9] 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.

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`	526	66 (13%)	5	0	5	36
`zordoz`	1380	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	141 (17%)	10	1	26	51
`gregor`	945	174 (18%)	13	2	42	142
`quadBG`	6780	220 (3%)	14	2	27	160
`quadMB`	6706	292 (4%)	16	2	29	174

Figure 6: 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 *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,[10] but do not change the size of a configuration lattice.

[10] This indirection did 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. Inlining does not affect overhead due to gradual typing, but greatly reduces the number of configurations.

5 Evaluating Typed Racket

5.1 Experimental Protocol

Section 5.2 and section 5.3 present the results of a comprehensive 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.
- For each configuration: recompile, run twice, and collect the results of the second run. Use the standard Racket compiler and runtime settings.
- Repeat the above steps N 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.[11] For the `quadBG` and `quadMB` benchmarks, the script utilized 30 of the machine's physical cores to collect data in parallel.[12] 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 and the appendix discusses the stability of individual measurements.

5.2 Evaluating Absolute Performance

Figures 7, 8, 9, and 10 present the results of measuring the benchmark programs in a series of overhead graphs. As in figure 5, the left column of 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 additionally include data for three versions of Racket released between June 2015 and February 2016. Data for version

[11] The Opteron is a NUMA architecture.

[12] The script invoked 30 green threads; these green threads invoked and monitored system processes to compile and run each configuration. The green threads pinned subprocesses to a fixed CPU core using the Linux `taskset` command.

6.2 are thin red curves with short dashes. Data for version 6.3 are mid-sized green curves 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 family 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 underlying 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 `lnm`, such configurations are a low proportion of the total.[13] The right endpoint shows how many configurations suffer over 20x performance overhead.[14] 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 graphs, 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 10), 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

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

[14] 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 graphs report an "over 20x" performance overhead for these configurations.

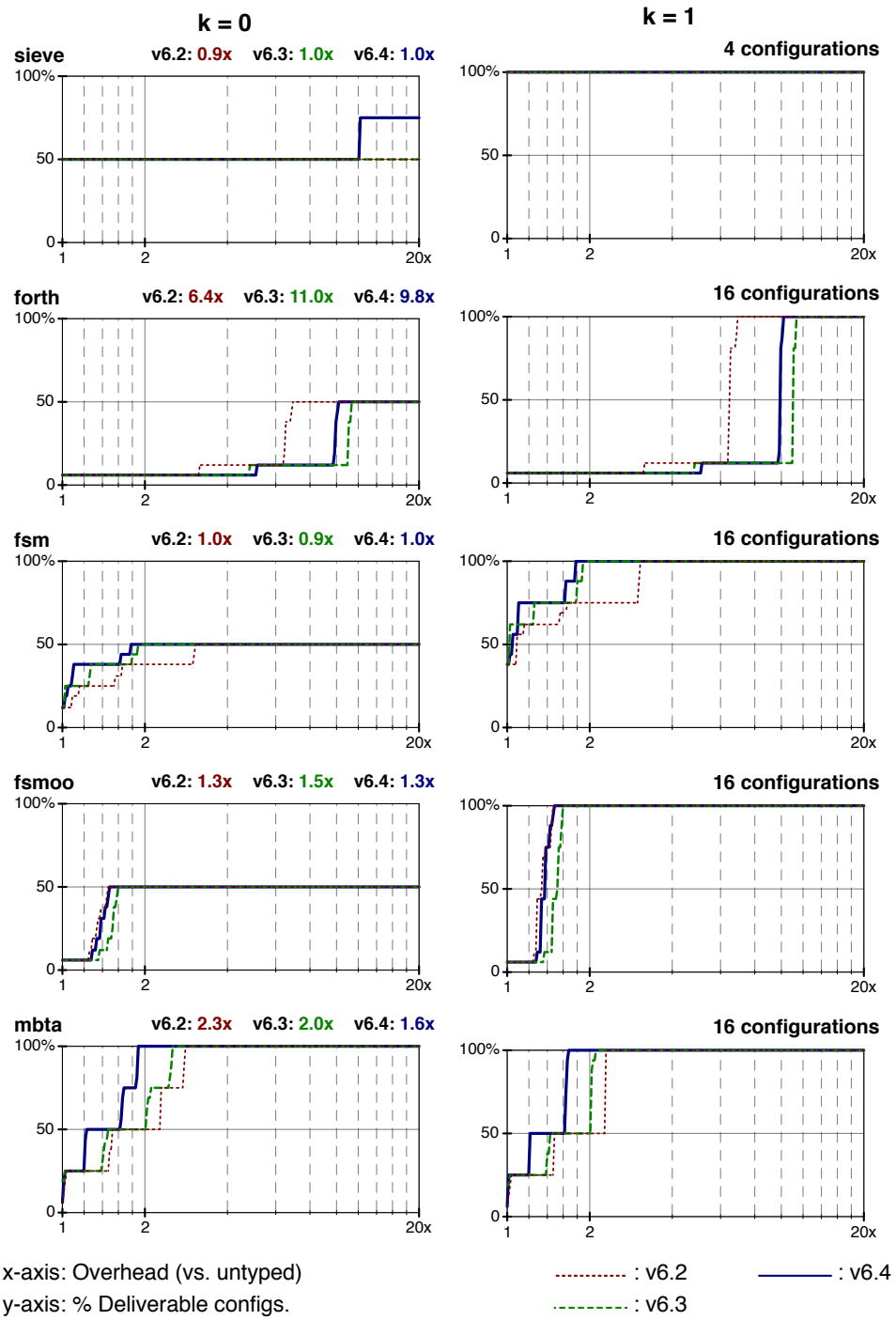


Figure 7: GTP overhead graphs (1/4)

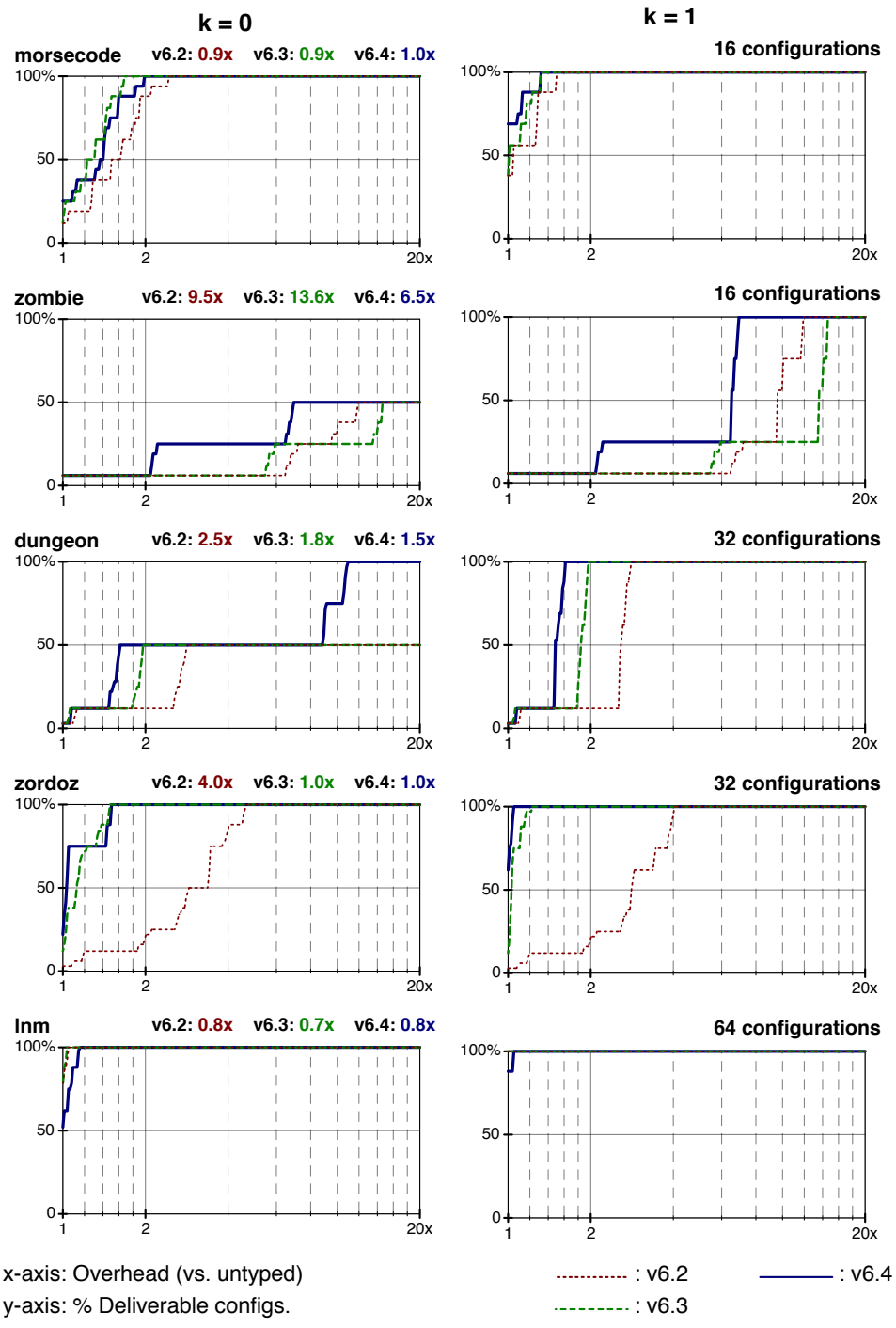


Figure 8: GTP overhead graphs (2/4)

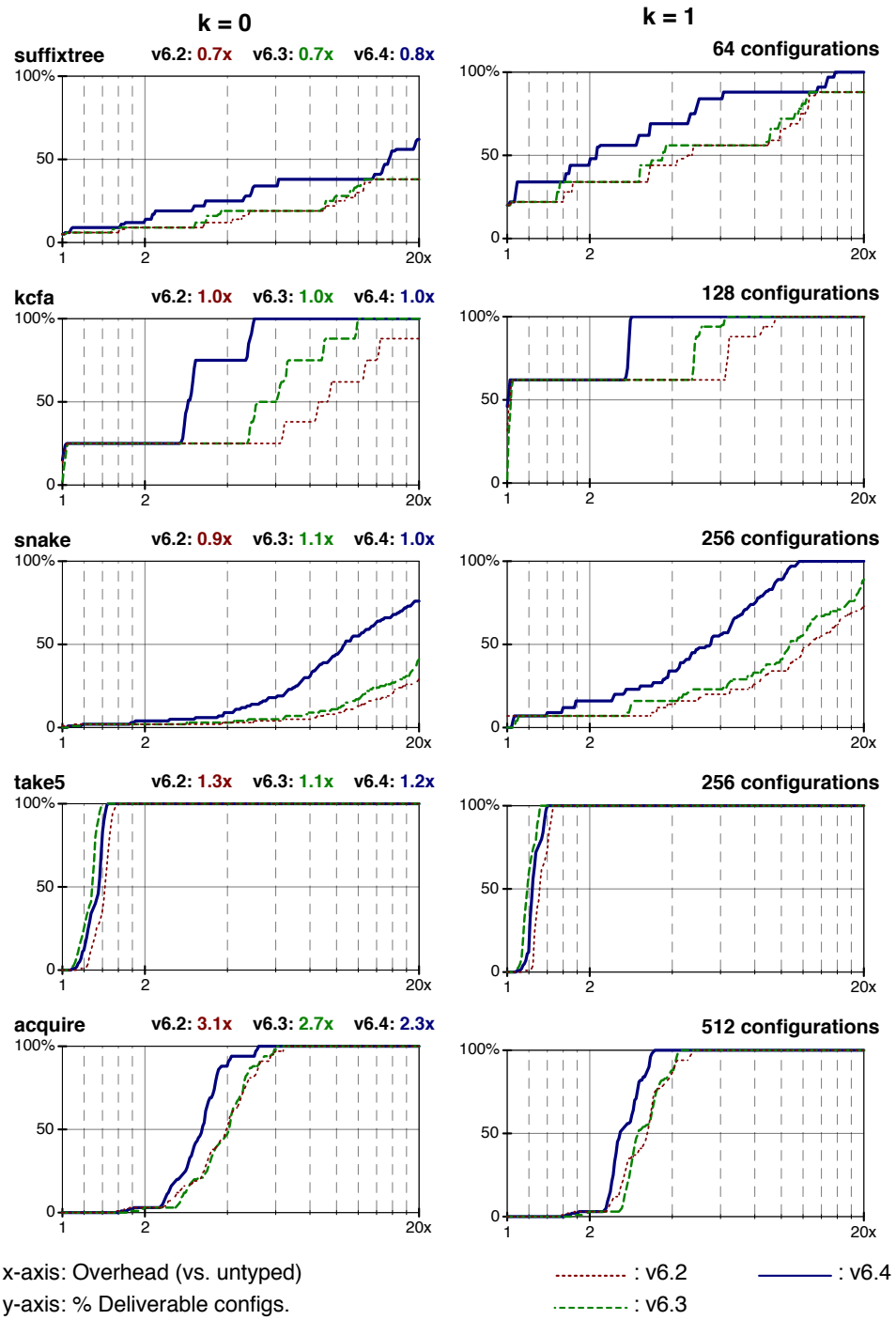


Figure 9: GTP overhead graphs (3/4)

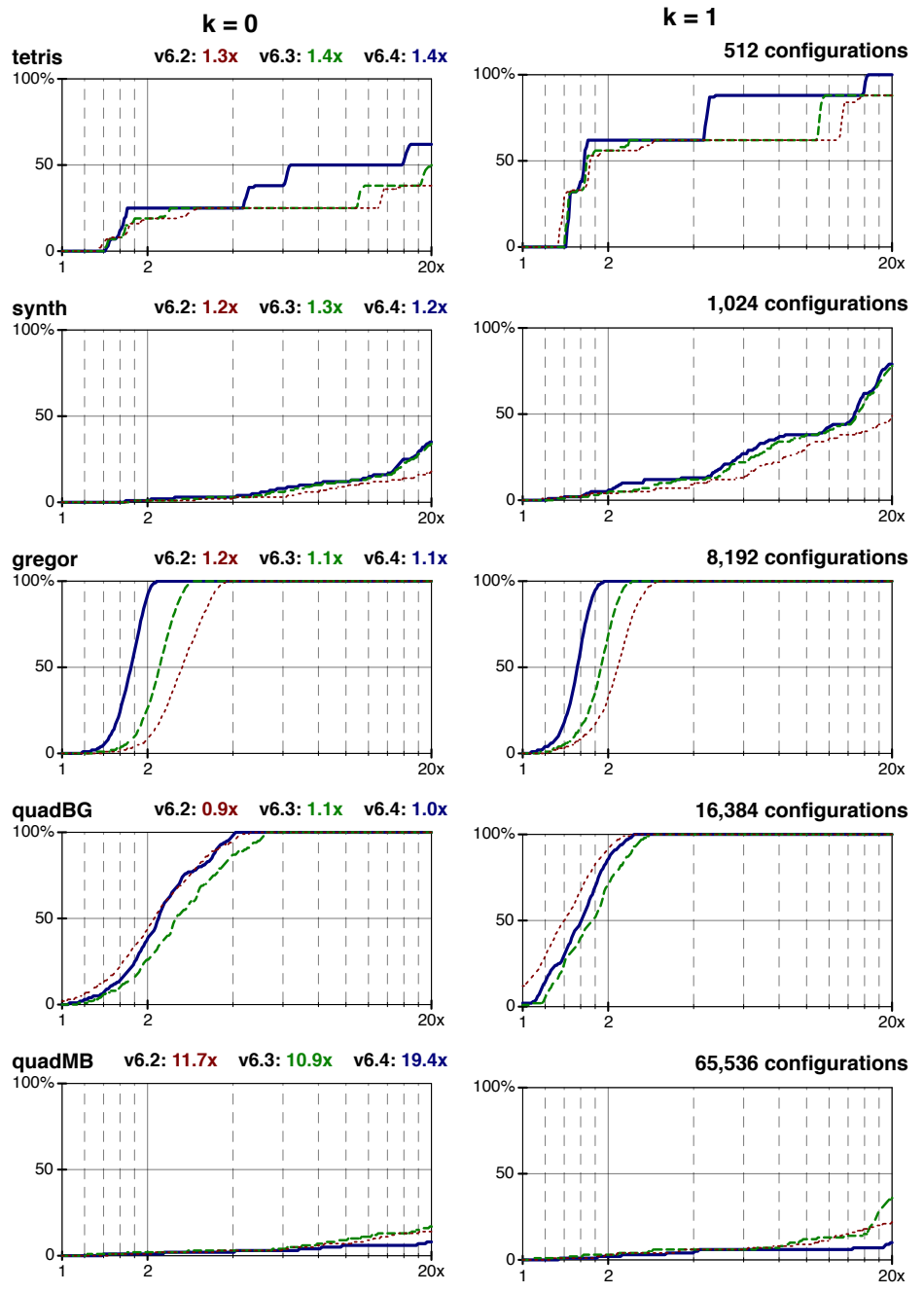


Figure 10: GTP overhead graphs (4/4)

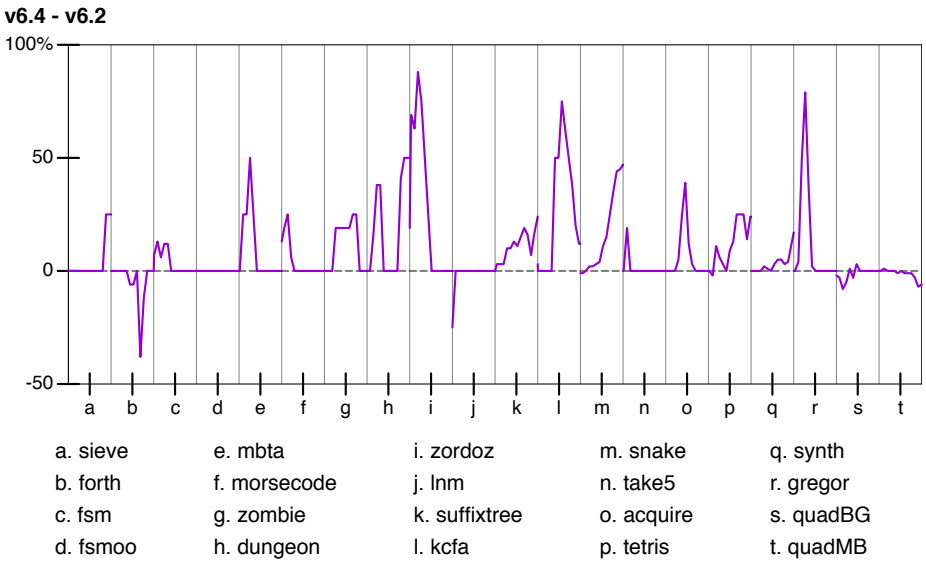


Figure 11: Relative performance of v6.4 versus v6.2

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.

The plot of figure 11 makes explicit how much version 6.4 is improved over version 6.2. It consists of twenty purple lines, one for each benchmark. These lines plot the difference between the curve for v6.4 and the curve for 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 improved performance in 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 typeable components. 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 modules. An evaluation of micro-level gradual typing would be severly limited; depending on the definition of a typeable component, such an evaluation might be limited to programs with 20 functions.

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 12 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 implicit suggestion of figure 12 is 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 13 provides evidence for the linear sampling suggestion of figure 12. It describes the eleven largest benchmarks in the GTP suite. The solid purple lines from figure 11 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. Again, the lesson is that 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 12. The details for figure 13 are analogous:

- To generate one random sample, select 80 configurations 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 12 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 13 the upper bound for the number of D-deliverable configurations on the right half of figure 12 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.

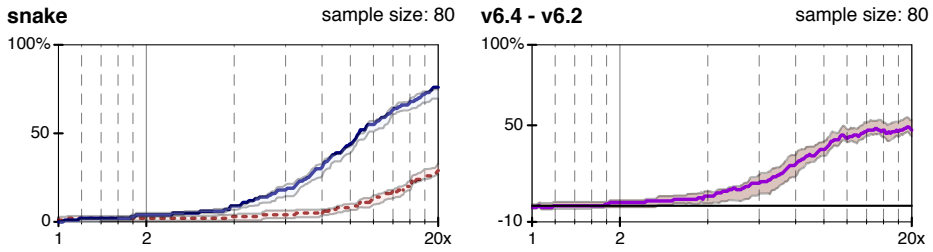


Figure 12: Approximating absolute performance

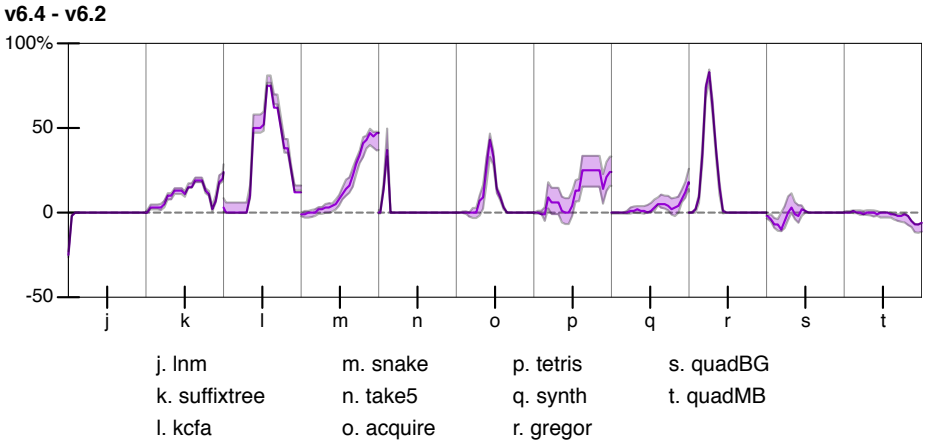


Figure 13: Approximating relative performance

7 Threats to Validity

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

There are four significant threats to the protocol. First, the benchmark programs are relatively small. Larger programs might avoid the pathological overheads in the benchmarks, though the results for `quadMB` and `synth` are evidence to the contrary.

Second, a few benchmarks have little data (less than 6 samples per configuration, details in the appendix) due to time limitations.[15] It is therefore possible that some samples are not truly representative.

Third, the configurations running in parallel reference the same Racket executable and external libraries. This cross-reference is a potential source of bias.

[15] Parallelizing the experiment would yield more samples, but would also add confounding variables to the measurements. See `http://prl.ccs.neu.edu/blog/2016/08/03/a-few-cores-too-many/` for one relevant anecdote.

Fourth, the Racket just-in-time compiler includes heuristic optimizations. The protocol of compiling *once* before collecting one sample does not control for these heuristics. 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 three 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. For example, `quadBG` and `quadMB` describe the same code with different types and have very different performance characteristics. Whereas all configurations of the former are 6-deliverable, only a small fraction of `quadMB` configurations are 20-deliverable.

Second, 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) may significantly reduce the overhead of gradual typing.

Finally, when the Racket contract system discovers a type boundary error, its type soundness theorem guarantees that programmers receive the exact type boundary, type annotation, and incompatible value in the error message. This blame assignment has practical value for developers (Dimoulas et al. 2012), 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, Reticulated Python's transient semantics checks the type of a mutable data structure when typed code reads from the structure, but not when untyped code writes to it, avoiding the need to proxy such data structures (Vitousek et al. 2014). StrongScript provides only nominal subtyping for objects, largely because structural subtyping incurs a higher runtime cost (Richards et al. 2015). The question is whether these alternatives are sufficiently practical.

8 Dissecting Performance Overhead

Our evaluation demonstrates that adding types to an arbitrarily chosen subset of Racket modules in a program can impose large performance overhead. This section explains with a few examples how such overheads may arise, both as inspiration for maintainers of gradual type systems and as anti-patterns for developers.

8.1 High-Frequency Typechecking

No matter the cost of a single runtime type check, if the check occurs frequently then the program suffers. The program in figure 14, for example, calls the typed function `stack-empty?` one million times from untyped code. Each call is type-correct; nevertheless,

```
#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 14: A high-frequency type boundary

Typed Racket validates the argument `stk` against the specification (`Listof A`) one million times. These checks dominate the performance of this example program, simply because many values flow across the module 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.

8.2 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 the 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.

Similarly, the Racket script in figure 15 executes in approximately twelve seconds. Changing its language to `#lang typed/racket` improves its performance to under 1 millisecond by removing a type boundary to the `trie` library. The drastic improvement is due to the elimination of the rather expensive dynamic check for the `trie` type.[16]

8.3 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-

[16] 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 15: 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 16: Accumulating proxies in `fsm`.

generated type boundary. These boundaries make it difficult to predict the overhead of gradual typing statically.

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.

8.4 Layered Proxies

Higher-order values that repeatedly flow across type boundaries may accumulate layers of type-checking proxies. These proxies add indirection and space overhead. Collapsing layers of proxies and pruning redundant proxies is an area of active research (Greenberg 2015; Herman et al. 2010; Siek and Wadler 2010).

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 16 and 17 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

```
#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 17: Accumulating proxies in `forth`.

```
#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 18: Adapted from the `zombie` 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.[17]

The `zombie` benchmark exhibits similar overhead due to higher-order functions. For example, the `Posn` datatype in figure 18 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.

[17] 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.

```
#lang typed/racket
(provide add-votes)

(define total-votes : Natural 0)

(: add-votes (Natural -> Void))
(define (add-votes n)
  (set! total-votes (+ total-votes n)))
```

Figure 19: Erasing types would compromise the invariant of `total-votes`.

8.5 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 thus runs significantly faster when fully typed.

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 *optionally typed* languages[18] [19] [20] [21] that provide static checks but sacrifice type soundness. Programs written in such languages run with zero overhead, but are suceptible to the hard-to-trace bugs and silent failures explained in section 2.2.

[18] `http://www.typescriptlang.org/`
[19] `http://hacklang.org/`
[20] `http://flowtype.org/`
[21] `http://mypy-lang.org/`

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.
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 graphs*. 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 graphs 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. 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).

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). High-cost types may be a symptom of inefficiencies in the translation from types to dynamic checks. Recent calculi for space-efficient contracts (Greenberg 2015; Siek et al. 2015) may provide insight for eliminating proxies. 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).

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). 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.

Joe Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007.

Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Fine-grained Language Composition: A Case Study. In *Proc. European Conference Object-Oriented Programming*, pp. 3:1–3:27, 2016.

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.

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.

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

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

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

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.

John Hughes. Why Functional Programming Matters. *The Computer Journal* 32(2), pp. 98–107, 1989.

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`

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, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of Gradual Type Inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 481–494, 2012.

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.

John C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *Proc. Information Processing*, 1983.

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, 2015.

Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the Design Space of Higher-Order Casts. In *Proc. European Symposium on Programming*, pp. 17–31, 2009.

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

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In *Proc. Summit on Advances in Programming Languages*, pp. 274–293, 2015.

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.

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.

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.

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.

Philip Wadler and Robert Bruce Findler. Well-typed Programs Can't be Blamed. In *Proc. European Symposium on Programming*, pp. 1–15, 2009.

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 contains links to some of the anecdotes that triggered this investigation into the performance of gradual typing. The online supplement to this paper includes copies of the email threads and documents cited below.

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`

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:

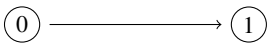
- the name and size of each module;
- whether each module has an adaptor;
- the number of identifiers imported and exported by the module;
- and a graph of inter-module dependencies, with edges from each module to the modules it imports from.

Modules are ordered alphabetically. Figure 3 uses this ordering on modules to represent configurations as black and white rectangles. For example, the node in figure 3 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 20 derives a natural number for each configuration using the alphabetical order of module names. Configuration 4 in figure 20 (binary: `0100`) is the configuration where only `main.rkt` is typed.

sieve

0. `main` 1. `streams`

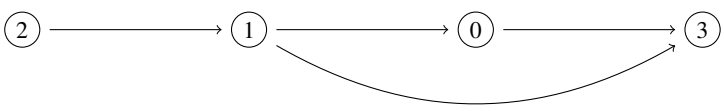
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	16	13		9	0
1	19	4		0	9



forth

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

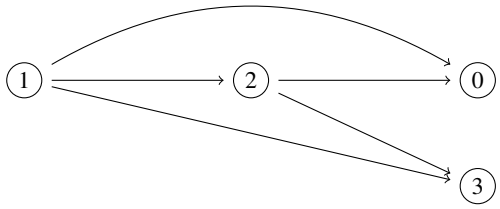
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	132	14		7	2
1	79	4		9	1
2	9	3		1	0
3	35	9		0	14



fsm

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

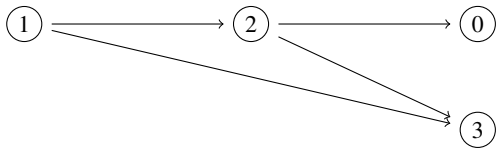
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	84	28	✓	0	20
1	24	10		17	0
2	46	12		13	4
3	28	6		0	6



fsmoo

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

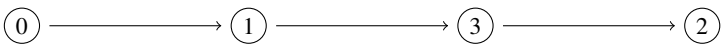
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	111	35	✓	0	5
1	18	11		4	0
2	42	28		8	1
3	23	9		0	6



mbta

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

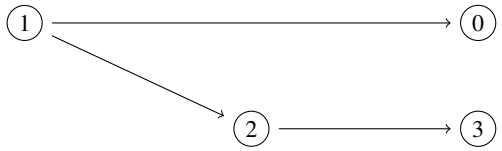
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	41	10		6	0
1	40	4		1	6
2	98	44		0	1
3	87	13		1	1



morsecode

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

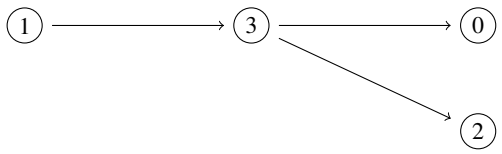
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	88	23		0	13
1	25	6		14	0
2	13	5		1	1
3	33	4		0	1



zombie

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

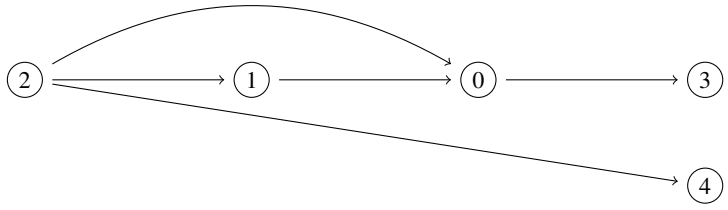
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	16	4	✓	0	7
1	38	9		3	0
2	12	6		0	5
3	236	8		12	3



dungeon

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

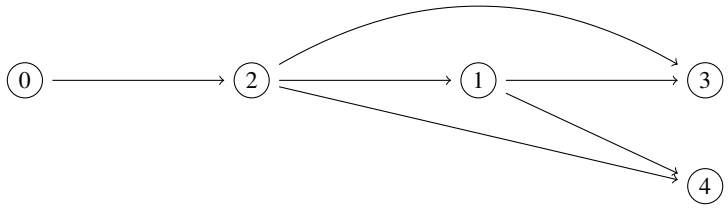
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	114	14		2	38
1	61	12		19	10
2	318	31		34	0
3	9	4		0	2
4	24	5		0	5



zordoz

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

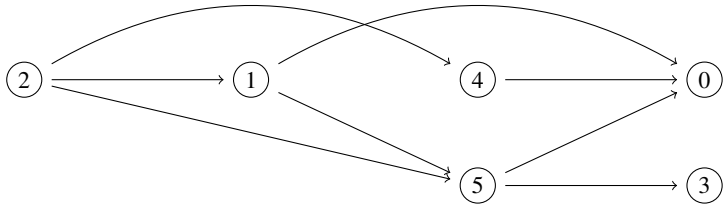
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	15	1		1	0
1	57	16		4	6
2	295	38		10	1
3	613	107		0	6
4	400	53		0	2



lnm

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

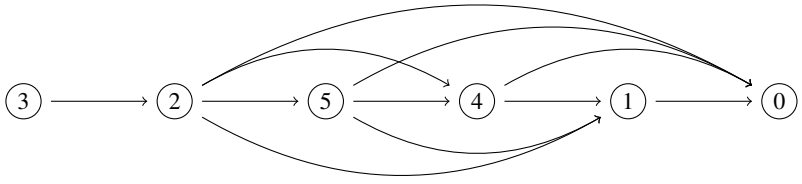
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	36	7		0	12
1	153	41		17	1
2	22	13		15	0
3	142	32	✓	0	9
4	38	8		4	1
5	97	13	✓	13	26



suffixtree

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

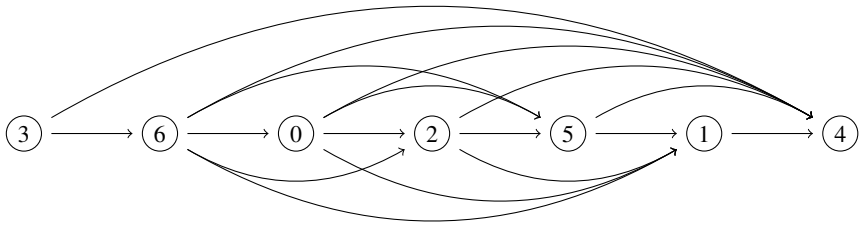
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	8	0	✓	0	108
1	119	40		27	66
2	110	11		66	3
3	13	2		3	0
4	101	40		49	20
5	186	36		59	7



kcfa

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

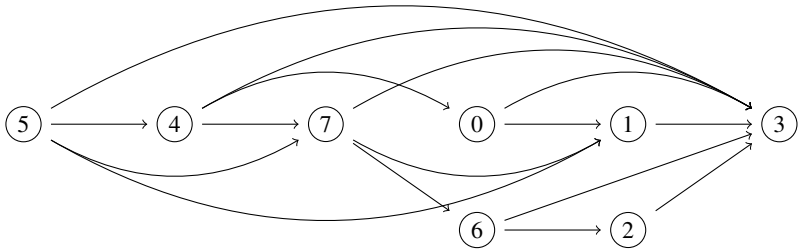
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	49	7		53	3
1	24	7	✓	21	56
2	38	10	✓	39	28
3	29	9		27	0
4	18	0	✓	0	126
5	20	6	✓	35	12
6	51	14		56	6



snake

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

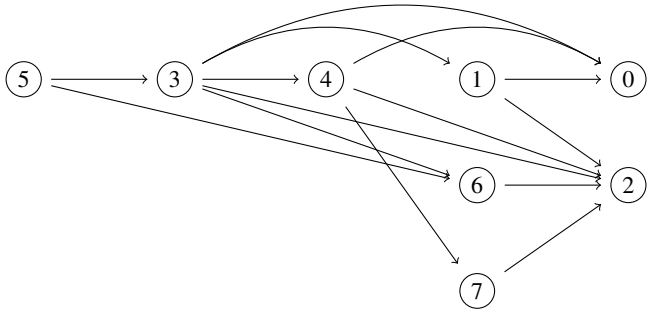
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	20	9		21	2
1	17	0		16	15
2	8	1		16	1
3	12	8	✓	0	112
4	16	6		21	2
5	33	10		26	0
6	31	12		23	6
7	23	5		17	2



take5

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

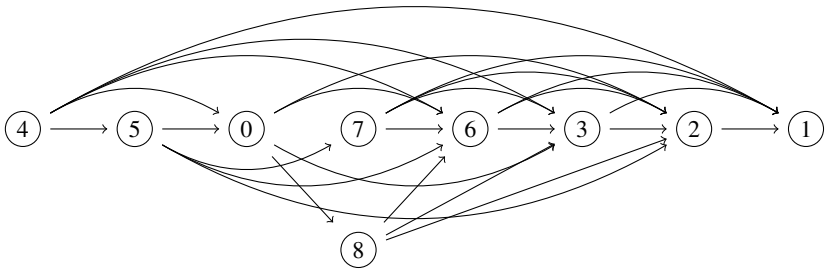
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	25	2		0	24
1	15	2	✓	0	35
2	32	3		15	1
3	101	11		19	1
4	71	4		20	1
5	33	0		3	0
6	27	7		7	4
7	23	-2		7	5



acquire

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

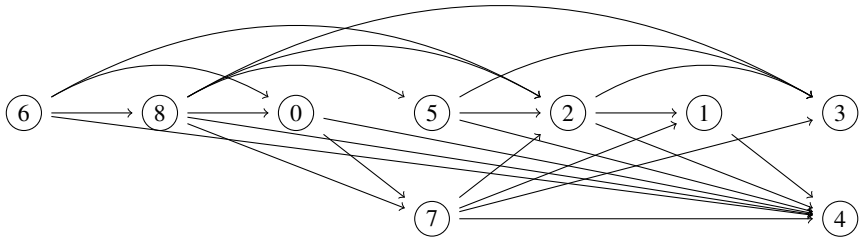
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	262	26		96	4
1	35	4		0	15
2	214	46		3	162
3	433	68	✓	30	150
4	33	7		72	0
5	71	11		65	3
6	359	81	✓	60	170
7	86	16		94	2
8	161	45	✓	91	5



tetris

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

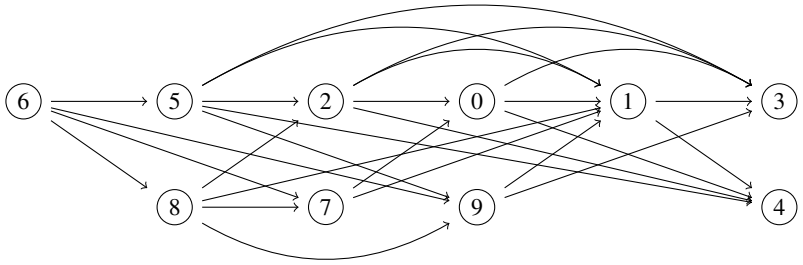
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	22	3		28	6
1	20	7		22	8
2	59	29		29	64
3	7	3		0	12
4	14	6	✓	0	154
5	14	8		41	1
6	21	13		44	0
7	36	10		45	12
8	53	28		51	3



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`

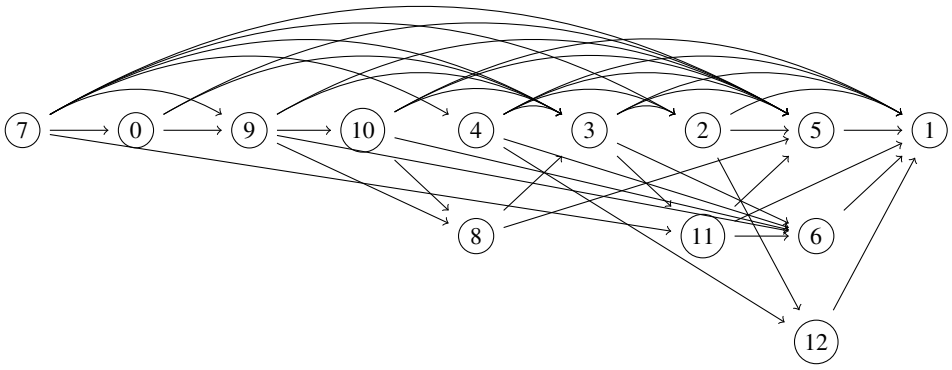
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	102	7		39	6
1	214	9		25	84
2	65	17		42	2
3	132	11		0	45
4	12	0	✓	0	64
5	53	22		44	1
6	94	11		8	0
7	61	17		17	2
8	32	28		20	2
9	70	19		23	12



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`

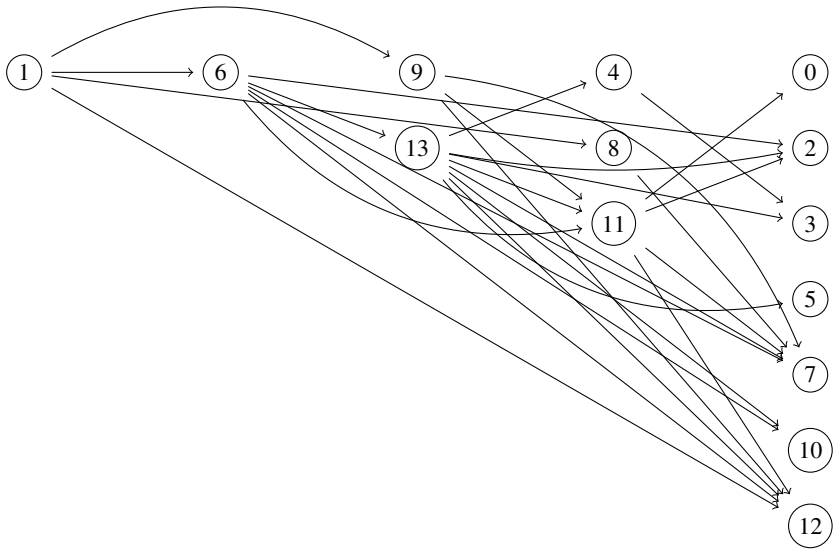
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	58	7		56	16
1	13	-1	✓	0	104
2	71	18		46	30
3	107	24		62	84
4	57	9		79	3
5	18	-1	✓	13	198
6	37	12		13	45
7	125	7		93	0
8	111	32		61	40
9	33	4		36	6
10	90	21		61	13
11	49	14		44	16
12	176	28		13	22



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`

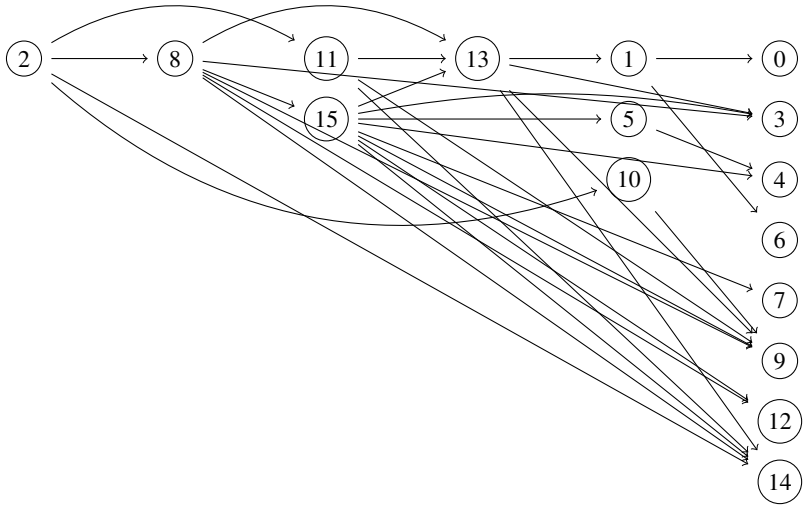
	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	5128	42		0	2
1	22	8		71	0
2	56	14		0	15
3	146	10		17	4
4	17	7	✓	0	34
5	6	0	✓	0	5
6	219	25		129	1
7	179	29		0	165
8	30	1		33	1
9	112	2		116	1
10	72	11		0	8
11	212	29		108	45
12	143	0		0	340
13	438	42		151	4



quadMB

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

	Untyped LOC	Ann. LOC	Adaptor?	# Imports	# Exports
0	3	1		0	1
1	187	45		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



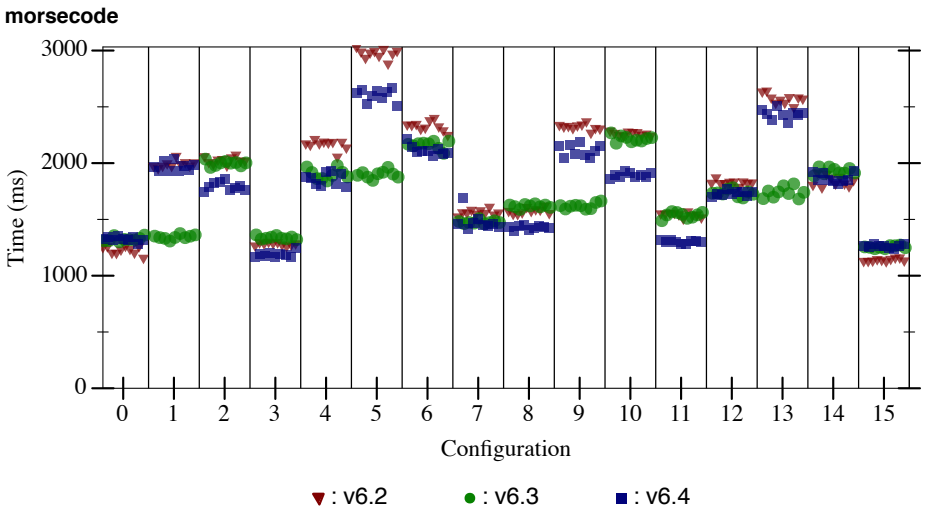


Figure 20: Exact running times in `morsecode`.

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 20 and 21 qualify this assumption.

Figure 20 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 the experiment recorded them.

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 21 plots their exact typed/untyped ratios on a logarithmic scale. Similar to figure 20, 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 21 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 21 provides some evidence that the average of a given sequence of typed/untyped ratios is an accurate representation of the true typed/untyped ratio.

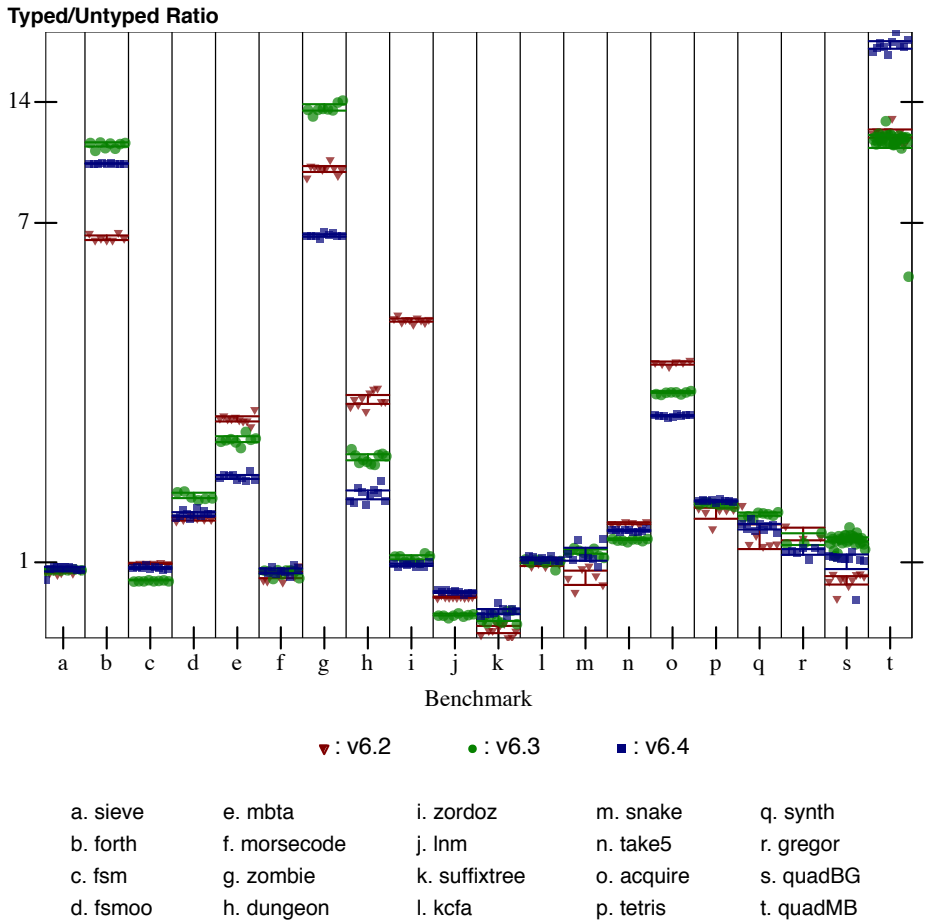


Figure 21: typed/untyped ratios, on a logarithmic scale.

Benchmark	v6.2	v6.3	v6.4	Benchmark	v6.2	v6.3	v6.4
`sieve`	10	6	10	`suffixtree`	10	8	9
`forth`	7	8	9	`kcfa`	6	8	9
`fsm`	6	8	9	`snake`	6	8	8
`fsmoo`	6	6	11	`take5`	6	8	9
`mbta`	10	8	9	`acquire`	6	8	9
`morsecode`	10	9	9	`tetris`	6	8	9
`zombie`	10	8	9	`synth`	6	8	9
`dungeon`	10	10	10	`gregor`	3	3	7
`zordoz`	10	8	9	`quadBG`	10	30	10
`lnm`	10	8	9	`quadMB`	10	30	10

Figure 22: Samples per benchmark

Benchmark	# Mod.	D = 1	D = 3	D = 5	D = 10	D = 20
`sieve`	2	0	0	0	0	0.5
`forth`	4	0	0	0	0	0
`fsm`	4	0	0	0	0	0
`fsmoo`	4	0	0	0	0	0
`mbta`	4	0	1	1	1	1
`morsecode`	4	0	1	1	1	1
`zombie`	4	0	0	0	0	0
`dungeon`	5	0	0	0	0.5	1
`zordoz`	5	0	1	1	1	1
`lnm`	6	0.17	1	1	1	1
`suffixtree`	6	0	0	0	0	0.17
`kcfa`	7	0	0.2	0.97	1	1
`snake`	8	0	0	0	0.08	0.5
`take5`	8	0	1	1	1	1
`acquire`	9	0	0.02	0.83	1	1
`tetris`	9	0	0	0	0.17	0.17
`synth`	10	0	0	0	0	0

Figure 23: Proportion of *D*-deliverable conversion paths.

10.4 Miscellaneous Figures

The table in figure 22 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 23 answers the hypothetical question of whether there exists any *performant conversion paths* through a performance lattice. More precisely, a *D*-deliverable *conversion path* in 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 space.

Figure 24 plots the average-case and worst-case overheads in the benchmark programs.

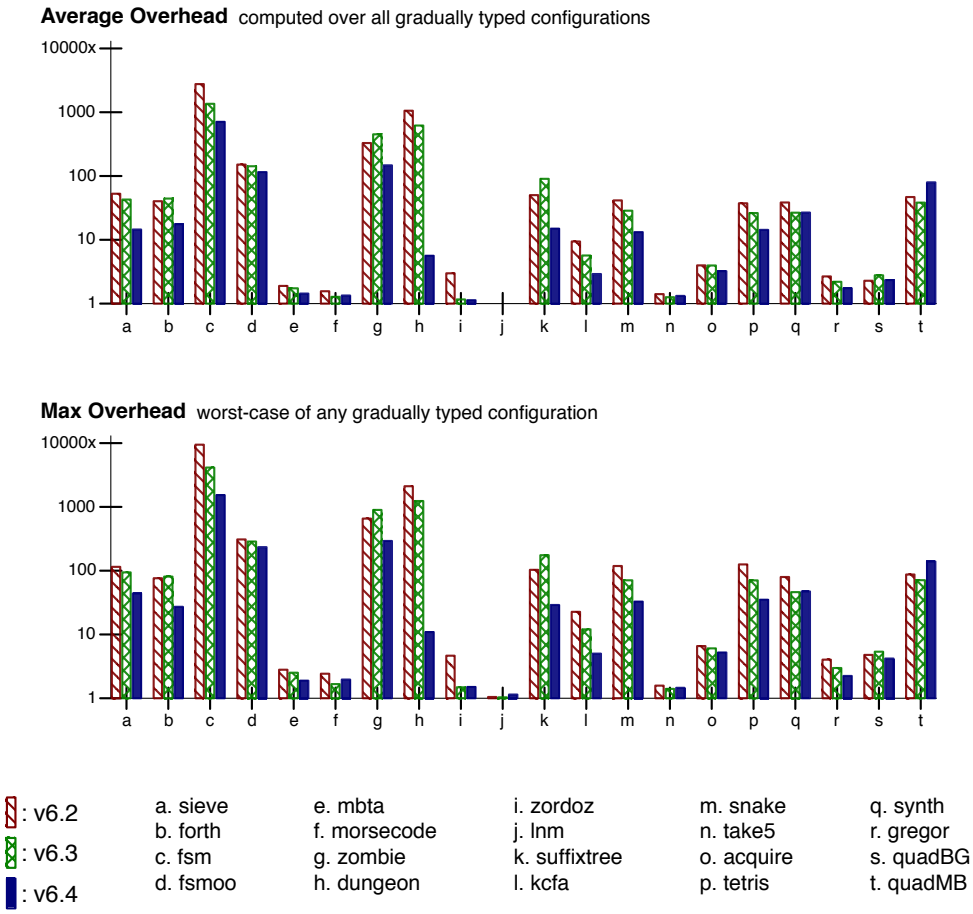


Figure 24: Average and worst-case overhead