# GTP Benchmarks for Gradual Typing Performance

Ben Greenman
Brown University
Providence, Rhode Island, USA
benjaminlgreenman@gmail.com

## ABSTRACT

Reproducible, rigorous experiments are key to effective computing research because they provide grounding and a way to measure progress. Gradual typing is an emerging area that desperately needs such grounding. A gradual language lets programmers add types to part of a codebase while leaving the rest untyped. The critical research question is how to balance the guarantees that types provide against the run-time cost of enforcing them. Either weaker guarantees or better implementation methods could lead to answers, but without benchmarks for reproducibility there is no sound way to evaluate competing designs.

The GTP Benchmark Suite is a rigorous testbed for gradual typing that supports reproducible experiments. Starting from a core suite of 21 programs drawn from a variety of applications, it enables the systematic exploration of over 40K gradually-typed program configurations via software for managing experiments and for analyzing results. Language designers have used the benchmarks to evaluate implementation strategies in at least seven major efforts since 2014. Furthermore, the benchmarks have proven useful for broader topics in gradual typing.

## CCS CONCEPTS

• **General and reference** → *Measurement*; *Performance*; • **Software and its engineering** → Compilers.

## KEYWORDS

reproducibility, benchmarks, performance, gradual typing

**ACM Reference Format:**
Ben Greenman. 2023. GTP Benchmarks for Gradual Typing Performance. In *2023 ACM Conference on Reproducibility and Replicability (ACM REP '23), June 27–29, 2023, Santa Cruz, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3589806.3600034

## 1 INTRODUCTION

Well-designed benchmarks that support reproducible experiments are key drivers for computing research [45, 154, 93]. Examples include the Gabriel benchmarks for LISP [39], DaCapo for Java [12], Scalabench for Scala [112], Renaissance for parallel Java [105], FPBench for floating-point kernels [24], ManyBugs and IntroClass for C program repair [81], the Magma fuzzing benchmark [64], `nlrpBENCH` for software requirements [140], `NELA` for news and misinformation [101, 68], Mälardalen WCET for worst-case time bounds [63], and B2T2, the Brown Benchmark for Table Types [83]. Without benchmarks and a method for reproducibility, it is all too easy for research to stagnate or go adrift [9, 141, 139, 120].

Gradual typing is one area that is at risk of veering off. Though originally envisioned to benefit working programmers [113, 142, 89, 60], the stack of theoretical designs that lack rigorous evaluation is growing year by year (e.g., [4, 86, 111, 85]). Closing the gap is an enormous task, however, because each design needs both a full implementation and a reproducible analysis.

Since their initial release in 2014, the GTP Benchmarks have supported rigorous and reproducible evaluation for gradual typing (section 2.3). The core of the suite is a set of 21 programs that represent a variety of practical tasks and, critically, send a variety of data across type boundaries (section 3). Toward the goal of *rigor*, each benchmark systematically supports configurations of typed and untyped code (section 4). A benchmark with $N$ modules has $2^N$ configurations, all of which must run efficiently for gradual typing to achieve an unqualified success. Toward *reproducibility*, companion software packages help to generate configurations, manage experiments, and visualize results (section 5). All the code is available on GitHub and Software Heritage [121, 122, 123]:

```
https://github.com/utahplt/gtp-benchmarks
https://github.com/utahplt/gtp-measure
https://github.com/utahplt/gtp-plot
```

The paper concludes with a report on lessons learned (section 6), related work (section 7) and a brief discussion (section 8).

*Goals.* This paper introduces a benchmark suite and explains how the suite supports reproducibility. For researchers in the area, our goal is to promote usage of, criticisms of, and extensions to the GTP Benchmarks. For everyone else, our goal is to encourage similar efforts in other domains by providing an example.

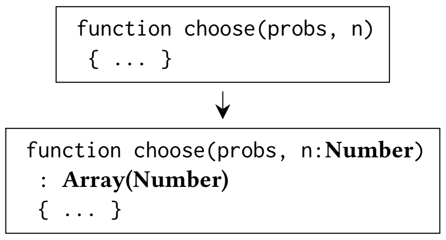
## 2 GRADUAL TYPING

One of the oldest questions in programming language design is whether to include a type system. In the early days, languages such as Cobol and Fortran used types to generate efficient machine code [110, 7, 104]. For example, the Fortran statement:

```
    integer,dimension(8) :: x
```

declares an array `x` with a type that specifies its size and contents. Meanwhile, Lisp [90] demonstrated the "power, flexibility, simplicity, and reliability" of untyped code (paraphrasing Hoare [67]), in which any variable can point to any sort of value at run-time.

Although some Lisp implementations permitted optional type annotations [94, 127], the typed and untyped styles developed over time into two different worlds. Typed languages came to provide strong guarantees, such as memory safety in Java [43], data-race



```
function choose(probs, n)
 { ... }
```

```
function choose(probs, n:Number)
 : Array(Number)
 { ... }
```

**Figure 1: Gradually adding types**

freedom in Rust [109], equational reasoning in Haskell [158, 42], and correctness proofs in Coq [137]. Untyped languages became the standard in web programming [44, 128], data science [136, 26], and exploratory work [82]. The language communities grew apart as well, with some experts hailing the "obvious" advantages of types [78] and others in strong opposition [66, 153].

Recent work in the area of gradual typing has begun to close the gap. The most prominent success to date is TypeScript, a language that lets JavaScript programmers enrich their code with types [146, 11]. Thousands of JavaScript apps have migrated to TypeScript [126, 25] because it offers three key affordances: (1) types may be added piece-by-piece to a JavaScript program; (2) types enhance developer tools, such as autocomplete; and (3) its programs can interoperate with existing JavaScript libraries. Point 1 is the main technical advance, and it was pioneered by gradual typing research [142, 113, 89, 60]. Figure 1 illustrates: the function `choose` starts off untyped, gains types for one parameter and for its return type (leaving one parameter untyped), and remains a valid target for callers in other TypeScript or JavaScript modules. For large codebases that need the organization that types provide (a growing pain experienced by Dropbox [71], Twitter [152], and even the creator of Python [69]), this ability to incrementally add types with minimal changes to existing code is much better than the alternative of porting to a different, typed language.

## 2.1 What Do Gradual Types Mean?

In general, the aim of a gradual type system is to combine the best aspects of typed and untyped code. But exactly what these best aspects are is subject to debate, especially when it comes to the types [55]. From the perspective of modern typed languages, soundness is key. Types must be reliable predictions about how a program behaves at runtime. If a variable `n` has type `Number`, for example, then the language must guarantee that `n` is inhabited only by numbers.

However, soundness guarantees do not hold in TypeScript—nor in many other gradual languages [54]. When a JavaScript program invokes the `choose` function in fig. 1, it can send any kind of input: a string, an array, or anything else. Unless the TypeScript programmer adds an explicit check for numbers, the invalid argument will propagate through the function body, at best raising an error and at worst silently producing wrong output. TypeScript types are therefore meaningless for debugging a flawed program or otherwise reasoning about behavior. They can detect issues only at compile-time, and only in typed code.

Research on sound gradual types, which do provide behavioral guarantees, has been ongoing for the past two decades. Theoretical work has established proof-of-concept designs for a spectacular variety of types [20, 31, 86, 115, 15, 99, 1, 100], and uncovered some negative results concerning the guarantees that gradual types can express [27, 28, 145, 77]. Implementations can be found in several languages [76, 160, 2], the most mature of which is Typed Racket [143, 144]. This research is exciting, but comes with a catch: runtime performance. Sound types require checks. Checks impose runtime costs, and these costs can slow down a correct program by several orders of magnitude [132, 59, 76, 58].

To get a sense of where high costs arise from, consider a call to the `choose` function (fig. 1):

```
  var nums = choose(x);
```

Sound gradual types guarantee that the input to `choose` is a number and that the result is an array of numbers. For the input, a quick tag check on `x` is enough. For the output, a more expensive check is needed because the array that `nums` points is a mutable data structure, thus future writes from untyped code may supply an invalid element (e.g., `nums[0] = "B"`). To enforce the full behavior of the type, `choose` must return a *wrapper* over the actual array.

Wrappers are an evident source of costs, as the runtime must allocate a wrapper and then redirect future operations. But the extent to which they slow down a program depends on how often these wrappers get created and used. Furthermore, wrappers are not the only source of costs. First-order checks on basic values and traversals of immutable data can get expensive too [59]. Thus, although theories for cheaper wrappers are important [117, 65, 116, 114, 47, 46] the critical questions are empirical: *which bottlenecks arise in practice* and *what can be done to avoid them?*

To summarize, gradual typing is poised to resolve a longstanding practical issue, but first it must find a way to reduce the costs of enforcing type guarantees. The way forward is clear: researchers must experiment with language designs and measure their performance. To facilitate progress, the community needs a comprehensive testbed for running and reproducing experiments.

## 2.2 What is a Gradual Typing Benchmark?

This paper contributes the GTP Benchmark Suite, which encompasses a collection of 21 programs (or rather, program families), an experimental method for gradual typing performance, and software for running experiments and visualizing results. Together, these three pieces enable rigorous, reproducible experiments.

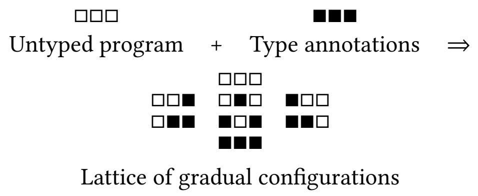
Each of the initial programs is a benchmark in the conventional sense (of, say, SPEC [125] or V8 [147]). As such, they barely scratch the surface of what becomes possible with gradual typing. The next step is to systematically explore *configurations* of typed and untyped code across the components in each program. In other words, each program gives rise to a family of related configurations, where each configuration is a runnable, partially-typed program. The thesis of our experimental method is that all configurations are important. The role of the support software is to create the configurations, manage experiments that test all configurations, and make sense of the exponentially-large result datasets.

A gradual typing benchmark is thus the family of configurations that arise from an initial codebase and a full set of type annotations.

```
function choose(probs, n)
```
: 0/3 types = □□□

```
function choose(probs, n:Number)
  : Array(Number)
```
: 2/3 types = □■■

**(a) Abstract representation of gradual code**



**(b) A program with three components has eight configurations**

**Figure 2: One benchmark, a family of programs**

For example, the two versions of the choose function from fig. 1 could be two configurations of a benchmark (fig. 2a). With a type for the first input to choose, there are eight total configurations, which fig. 2a presents in a lattice to show how they systematically explore all combinations of typed and untyped code obtainable by removing type annotations.[1]

The pressing question for a benchmark is how much *overhead* types add (in the form of runtime checks) relative to the completely untyped program configuration. Untyped performance is what the programmer can achieve without any gradual typing. Thoughout this paper, remarks about overhead (e.g., "a 3x slowdown") are all relative to the untyped configuration.

The GTP Benchmarks are written in Racket [103] and the families explore types at a module-level granularity. Racket is a natural fit because it ships with Typed Racket, a mature implementation of sound gradual typing that has been refined over 15+ years and supports a wide array of descriptive types, including types for continuations, first-class classes, and variable-arity functions [144, 131, 129, 134]. The module-level granularity means that a program with three modules leads to eight configurations; by contrast, fig. 2 used a mere three variables to generate eight configurations. Using a coarser granularity allows for an in-depth analysis of larger programs. Scaling is still an issue, but linear sampling can effectively draw approximate conclusions [58, 59].

## 2.3 Significance of the GTP Benchmarks

Sound gradual typing enables the creation of programs that mix typed and untyped code, but leaves open the question of how fast mixed programs run. The GTP Benchmarks were created to answer this question in the context of Typed Racket. Earlier work on gradual typing performance is sparse. A few papers present case studies [143, 2, 155, 3]. Others report on the performance of fully-typed code relative to untyped code, but not on the landscape of gradual configurations in between [106, 157]

[1]When adding types to untyped code, there may be an unlimited number of choices. For example, an identity function ($\lambda x.\ x$) matches an infinite number of simple types: $(\alpha \rightarrow \alpha)$, $((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$, . . .

Initial results were dismal, raising the question of whether sound gradual typing was a dead end as a research area [132]. Overheads exceeding 20x were the common case rather than pathologies. The question is still relevant today, which, as Greenberg [48] notes, puts pure-theory gradual typing research on shaky ground. But thanks to the reproducibility provided by the GTP Benchmarks, several research teams developed techniques and measured improvements:

- Changes to the implementation of wrapper values and the compilation of types contributed speedups within one year of the initial results [59].
- Collapsible function and array wrappers led to order-of-magnitude improvements for two benchmarks [38]. A collapsible, or space-efficient, wrapper drops redundant layers without loss of debugging information [47]. Collapsing additional contracts is a promising direction for future work.
- An alternative semantics for types, inspired by Reticulated Python [157], gave programmers a choice between fully-reliable types and faster checks [57, 50]. With the faster checks, the common case is under 10x; only one benchmark has any configurations with a higher slowdown.
- Pycket demonstrated that tracing JIT technology built upon PyPy [14], can reduce costs across the board without changing behavior [10].
- Corpse Reviver is a static analysis of untyped code relative to its type constraints. On the 12 GTP Benchmarks that do not rely on object-oriented features, the analysis reduces gradual typing overhead to a 1.5x maximum [95].

Outside of Typed Racket, adaptations of the GTP Benchmarks exist in several languages. Reticulated Python used the benchmarks to measure the cost of its weakly-sound types [58] and the benefits of two optimization techniques [156]. Grift used select benchmarks to validate a machine-tailored, collapsible implementation of wrappers [76]. Nom [96, 97] and HiggsCheck [107] used select benchmarks to test a redesign of gradual typing in which a value must be initialized with a correct label to enter typed code [162, 106, 96]. Although this design narrows the scope of gradually-typed programs, it has been adopted in two industry languages: Dart and Static Python [135, 84].

Looking beyond performance, the benchmarks have enabled reproducible experiments for other topics related to gradual typing. Phipps-Costin et al. [102] investigate the use of an SMT solver to find types for untyped code; the Grift benchmarks provide support. Lazarek et al. [79, 80] use the GTP Benchmarks to compare the effectiveness of type-driven debugging on strong, weak, and unreliable types. Their results agree with earlier theoretical work [56], but by only a slim margin, which underscores the importance of experiments. Recent work compares the ability of profiling tools to discover fast configurations.

## 3 THE BENCHMARKS

Any benchmark suite supports a basic form of reproducibility, since researchers can run the benchmarks on different systems and compare the results. Nevertheless, the first question a benchmark suite must answer is whether it is relevant to the problem at hand. Gradual typing is a technique for general-purpose programs and it suffers when typed and untyped code interact. Relevant benchmarks should explore a variety of type boundaries in practical applications.

Table 1 summarizes the 21 GTP Benchmarks. With the exception of sieve, a tiny program with one high-traffic module boundary, each is derived from an independent, useful program. The first columns of table 1 cite the source code (when available) and briefly states their original purpose. One distinction to bear in mind: the *HTDP games* were created to teach programming to beginners, in the style of *How to Design Programs* [36]; whereas the *games* were client/server applications. Further details on the origins of each benchmark are online in the benchmark documentation [61].

Most benchmarks originated as untyped programs. Only the few marked in the T Init column, such as take5, came with types written by their original author. The benchmarks incorporate such types when available and otherwise choose idiomatic types that enable a full lattice (fig. 2b) of configurations. Section 4 explains the challenge of finding suitable types.

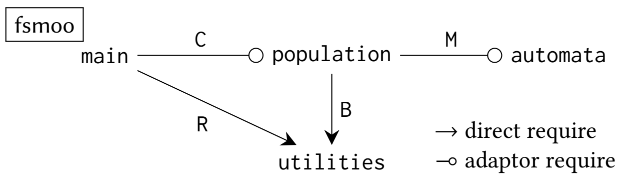
To a first approximation, the number of configurations in a benchmark is determined by the number of modules in the original program. There is, however a distinction between library modules that a program depends on and modules that are part of the program itself. In practice, adding types to a library module is generally less feasible than adding types to a module by the program's author. Each benchmark therefore has some *contextual modules* that remain fixed for all configurations, and some *migratable modules* to which types can be added or removed. In table 1, the columns T Lib and U Lib show a filled circle if there are any typed or untyped contextual modules (aside from base Racket libraries).

Each configuration uses the same underlying code; the only difference between configurations is their type annotations. This code is similar to that of the original program, but rarely identical. Section 4 explains why changes may be needed to run all configurations. One notable source of changes is that Racket struct types require a level of indirection that we call *adaptor modules* (Adapt). Adaptor modules change the structure of a program but not its performance; section 4.1 explains in more detail.

As a whole, the benchmarks exercise a variety of gradual typing behaviors. The remaining columns of table 1 display a filled circle (●) when a notable type is present at some boundary and an empty circle (○) otherwise. The columns stand for higher-order functions (HOF), polymorphic functions (Poly), recursive types (Rec), mutable data such as arrays or hashes (Mut), immutable data such as lists (Imm), first-class objects (Obj), and first-class classes (Cls).

## 3.1 Static Characteristics

Table 2 summarizes the migratable modules in the benchmarks. The first three columns are about size. They report the number of migratable modules (# Mod), the total lines of code in the untyped configuration (U LOC), and the additional lines in the fully-typed configuration (Ann LOC). For a program with $M$ modules, there are $2^M$ configurations; thus the benchmarks have between 4 and 16,384 configurations with a median of 64 (= $2^6$). The final two columns describe inter-module dependencies: the number of import/export links among migratable modules (# Bnd), and the number of identifiers that flow across these links (# Exp). Identifiers include value definitions and type definitions, but only the value definitions incur a cost at runtime.



Boundary types:
```
  B : Natural -> Population
  C : [Probability] Natural Op(Real) -> [Natural]
  M : Natural -> Automaton
  R : [Real] Real -> Real
```
Notation:
`[T]` is a homogeneous list of `T`'s
`Op(T)` is either a `T` or false

**Figure 3: Modules, dependencies, and boundary types**

*Example:* Figure 3 presents an overview of one benchmark, fsmoo. The purpose of this benchmark is to model competition in an economy. It is implemented using objects to represent finite-state automata (as opposed to the fsm benchmark, which uses mutable arrays instead). The program has four migratable modules: `main` drives the simulation, `population` models an economy of agents, `automata` models an individual agent, and `utilities` provides helper functions. There are no contextual modules. Each of the four module boundaries (represented as arrows) transmits one function type. The bottom half of the figure lists these types. Similar figures for the other benchmarks are in the benchmark documentation [61].

From this static picture alone, it is impossible to predict the runtime overhead in fsmoo. Every boundary could be expensive if a large number of functions crosses it, or if these functions are called repeatedly. In fact, the real source of overhead in fsmoo is not the functions but rather the types `Population` and `Automaton`, which represent objects. The methods of these objects get called repeatedly, leading to layers of wrappers on the inputs and results.

## 3.2 Dynamics: Higher-Order Wrappers

Table 3 reports on the uses of higher-order wrappers in the benchmarks. The main takeaway is that the suite exercises a variety of wrappers, as shown by the large numbers that appear in each column. Digging further requires some context.

First of all, the data in table 3 is specific to one pathological configuration, the *TWC configuration* (typed worst case), rather than an aggregate over the full $2^M$ lattice, which would be difficult to interpret. In the TWC configuration, every module is typed and imposes checks on its imports. This configuration does not appear in a normal lattice because typed-to-typed boundaries do not impose checks by default. It has the benefit that every boundary between migratable modules imposes a cost. However, the Typed Racket optimizer removes checks that typed code is guaranteed to satisfy. This optimization results in low numbers for sieve and take5.

**Table 1: Benchmarks overview: purpose and characteristics**

| Benchmark | Purpose | T Init | U Lib | T Lib | Adapt | HOF | Poly | Rec | Mut | Imm | Obj | Cls |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sieve | *prime generator* | ○ | ○ | ○ | ● | ○ | ○ | ● | ○ | ● | ○ | ○ |
| forth | *Forth interpreter* [51] | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ● | ● | ● |
| fsm | *economy simulation* [33] | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ |
| fsmoo | *economy simulation* [34] | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● | ○ |
| mbta | *subway map* | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ |
| morsecode | *Morse code trainer* [23, 148] | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ |
| zombie | *HTDP game* [151] | ○ | ○ | ○ | ● | ● | ○ | ● | ○ | ● | ○ | ○ |
| zordoz | *bytecode tools* [53] | ○ | ● | ○ | ● | ● | ○ | ● | ● | ● | ○ | ○ |
| dungeon | *maze generator* | ○ | ○ | ○ | ○ | ● | ● | ● | ● | ● | ● | ● |
| jpeg | *image tools* [161] | ● | ● | ● | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ |
| lnm | *data analysis* [52] | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ● | ○ | ○ |
| suffixtree | *string tools* [163] | ○ | ○ | ○ | ● | ● | ○ | ● | ● | ○ | ○ | ○ |
| kcfa | *program analysis* [92] | ○ | ○ | ○ | ● | ● | ○ | ● | ● | ● | ○ | ○ |
| snake | *HTDP game* [149] | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ | ○ |
| take5 | *game* [35] | ● | ○ | ○ | ● | ● | ○ | ○ | ○ | ● | ● | ● |
| tetris | *HTDP game* [150] | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ | ○ |
| acquire | *game* [32] | ● | ○ | ○ | ● | ● | ● | ● | ○ | ● | ● | ○ |
| synth | *music maker* [5] | ● | ○ | ○ | ● | ● | ○ | ○ | ● | ● | ○ | ○ |
| gregor | *time & date tools* [164] | ○ | ● | ○ | ● | ○ | ○ | ○ | ○ | ● | ○ | ○ |
| quadT | *typesetter* [16] | ● | ● | ○ | ● | ○ | ● | ● | ● | ● | ○ | ● |
| quadU | *typesetter* [17] | ○ | ● | ○ | ● | ○ | ● | ○ | ● | ● | ○ | ● |

**Table 2: Benchmark size and complexity**

| Benchmark | # Mod | U LOC | Ann LOC | # Bnd | # Exp |
|---|---|---|---|---|---|
| sieve | 2 | 35 | 17 | 1 | 9 |
| forth | 4 | 257 | 31 | 4 | 10 |
| fsm | 4 | 182 | 56 | 5 | 17 |
| fsmoo | 4 | 194 | 83 | 4 | 9 |
| mbta | 4 | 266 | 71 | 3 | 8 |
| morsecode | 4 | 159 | 38 | 3 | 15 |
| zombie | 4 | 300 | 25 | 3 | 15 |
| zordoz | 5 | 1,393 | 223 | 6 | 12 |
| dungeon | 5 | 541 | 69 | 5 | 38 |
| jpeg | 5 | 1,432 | 165 | 5 | 50 |
| lnm | 6 | 488 | 114 | 8 | 28 |
| suffixtree | 6 | 537 | 130 | 11 | 69 |
| kcfa | 7 | 230 | 53 | 17 | 62 |
| snake | 8 | 159 | 50 | 16 | 31 |
| take5 | 8 | 318 | 34 | 14 | 26 |
| tetris | 9 | 249 | 107 | 23 | 58 |
| acquire | 9 | 1,654 | 304 | 26 | 106 |
| synth | 10 | 837 | 138 | 26 | 51 |
| gregor | 13 | 945 | 175 | 42 | 142 |
| quadT | 14 | 6,685 | 307 | 27 | 174 |
| quadU | 14 | 6,779 | 221 | 27 | 160 |

Second, wrappers in Racket exist only for primitive values [130]. Objects are not primitive; instead, they are implemented with structs. Thus there is no column for object wrappers, and acquire and fsmoo (which use objects) have high numbers for structs.

Third, the table counts only applications (apps); that is, uses of wrapped values. Two other important statistics are the number of wrapper creations and the maximum depth of wrappers around any one value. Appendix A reports these details.

Moving on to the table, several observations are apparent: zombie, suffixtree, snake, and synth focus on procedures; forth, and fsmoo use the most structs; and synth makes a huge number of arrays. The object-oriented benchmarks fsmoo, take5, and acquire all have high numbers for structs and procedures. All but a few benchmarks have thousands of applications in at least one category, meaning they put gradual types to work.

## 3.3 Dynamics: Garbage Collection

Table 4 reports garbage collection details for the TWC configuration of each benchmark. On the whole, every benchmark allocates a significant amount of memory in the this configuration (at least 70MB) and some spend a considerable amount of time collecting garbage (seven spend 10 % or more).

The same caveats mentioned above (section 3.2) apply to the TWC data in this section. On one hand, it may spend far more time in garbage collection than any configuration in the lattice. On the other hand, it may spend too little time in collection because of the contract optimizer.

The columns in table 4 report four items: the percent of benchmark time spent on garbage collection (Total GC %), the average time spent per garbage collection (ms per GC), the maximum amount of data that was in use at the start of any one garbage collection (peak MB), and the percent difference between peak MB in the TWC configuration versus the untyped configuration. For three benchmarks, peak MB before a collection is lower than in

**Table 3: Usage of wrapped values (TWC Configuration)**

| Benchmark | Procedure apps | Struct apps | Array apps |
|---|---|---|---|
| sieve | 5 | 10 | 0 |
| forth | 152,149 | 51,801,426 | 0 |
| fsm | 1,010 | 2,524 | 3,305,323 |
| fsmoo | 2,300,769 | 28,285,389 | 17,952,160 |
| mbta | 498,809 | 949,909 | 0 |
| morsecode | 3 | 10 | 0 |
| zombie | 536,110 | 10 | 0 |
| zordoz | 572,191 | 317,828 | 88 |
| dungeon | 723,105 | 15,789,250 | 1,221,200 |
| jpeg | 24 | 25 | 5,899,946 |
| lnm | 1,908 | 30,703 | 0 |
| suffixtree | 3,935,912 | 10 | 108,592 |
| kcfa | 3,584 | 10 | 0 |
| snake | 11,739,420 | 22 | 0 |
| take5 | 2,717,007 | 25,940,019 | 0 |
| tetris | 5,071 | 22 | 0 |
| acquire | 2,098,839 | 8,634,064 | 0 |
| synth | 30,332,600 | 352 | 43,593,686 |
| gregor | 281 | 207 | 116,980 |
| quadT | 72,589 | 116,335 | 125,168 |
| quadU | 73,949 | 115,996 | 124,734 |

| Key: | $[0, 10^3)$ | $[10^3, 10^6)$ | $[10^6, 10^9)$ | $[10^9, \text{inf})$ |
|---|---|---|---|---|
| | white | yellow | orange | red |

**Table 4: Garbage collection info (TWC Configuration)**

| Benchmark | Total GC % | ms per GC | Peak MB | (vs U) |
|---|---|---|---|---|
| sieve | 16 % | 5 | 111.71 | (47 %) |
| forth | 3 % | 2 | 109.32 | (225 %) |
| fsm | 15 % | 12 | 88.71 | (7 %) |
| fsmoo | 14 % | 17 | 142.47 | (179 %) |
| mbta | 9 % | 7 | 93.24 | (3 %) |
| morsecode | 2 % | 1 | 74.10 | (−15 %) |
| zombie | 3 % | 2 | 74.00 | (94 %) |
| zordoz | 6 % | 3 | 91.85 | (2 %) |
| dungeon | 5 % | 3 | 138.64 | (67 %) |
| jpeg | 10 % | 6 | 88.66 | (2 %) |
| lnm | 23 % | 18 | 157.48 | (10 %) |
| suffixtree | 0 % | 0.4 | 88.19 | (277 %) |
| kcfa | 2 % | 2 | 75.55 | (−14 %) |
| snake | 1 % | 1 | 87.86 | (22 %) |
| take5 | 2 % | 1 | 105.59 | (21 %) |
| tetris | 2 % | 0.7 | 88.25 | (73 %) |
| acquire | 1 % | 2 | 135.73 | (49 %) |
| synth | 7 % | 5 | 113.80 | (17 %) |
| gregor | 14 % | 9 | 79.37 | (−10 %) |
| quadT | 16 % | 18 | 136.87 | (23 %) |
| quadU | 4 % | 5 | 171.81 | (69 %) |

the untyped configuration: `morsecode`, `kcfa`, and `gregor`. These benchmarks also have low wrapper use in table 3.

## 3.4 Version History

The GTP Benchmarks come with a semantic version number (current version: 9.2). Versioning lets the benchmarks improve while supporting the reproducibility of prior work. New experiments should use the latest release; reproduction studies should use the release noted in prior work (or a contemporary one). Below, we list major changes. More information is in the documentation and on the GitHub releases page [61, 62].

**Version 1** replaced two benchmarks, quadBG and quadMB, with two others that are better-suited to measure gradual typing overhead: quadU and quadT.

The quad benchmarks came from two programs by the same author: one untyped, one typed. Originally, quadBG and quadMB combined code from these programs and used different choices of type annotations. This was a poor choice because the typed codebase from the original author did more than add types; it also added structural constraints to data structures. The revised quadU is based on the untyped code and quadT on the typed code.

**Version 5** removed an unused call to format in the typed version of zordoz. Because the call happened only in typed code, it made the cost of gradual typing appear much too high.

**Version 9** improved acquire and take5 to do more meaningful work. Both benchmarks run games using AI players, and both were flawed. In acquire, every player eventually performed an illegal move and got kicked from the game. In take5, the driver script created a list of players that the game engine ignored in favor of its own, internally-made players. The updated acquire creates players that make only valid moves and the updated take5 uses the input list of players. These changes had negligible impact on gradual typing overhead.

## 4 TYPING ALL CONFIGURATIONS

A rigorous assessment of gradual typing performance must consider all configurations a lattice in order to test the promise of gradual typing. The GTP Benchmarks support the reproduction of all configurations from two copies of the benchmark source code (fully-typed and untyped). In part, this reproducibility is due to a library that conditionally installs type boundary checks [124]. Finding types for untyped code that work for all gradual configurations can, however, be difficult.

First, the type checker may require casts or refactorings to deal with untyped code. For example, untyped Racket code may assume that the application `(string->number "42")` returns an integer; though correct, the type checker cannot follow the value-dependent reasoning involved.

Second, limitations of the type system may require code changes. An interesting example comes from Python, which has a flexible `range` function that can return a variety of iterators depending on the input. Reticulated, a gradual type system for Python, thus uses the dynamic type to describe the output [157, 57]. For programs that expect a simple list of integers as the result, it may be preferable to replace the use of `range` to enable a more-detailed type.

Third, some type boundaries may lack run-time support. Typed Racket cannot enforce the type `(U (-> Real) (-> Integer))` at a boundary because its contracts lack unions for higher-order wrappers. The work-around is to rewrite the boundaries or, if possible, simplify the types. For the above, `(-> Real)` works as a simplified type. The zombie benchmark contains an example of the latter fix. Several functions in zombie implement a message-passing protocol where they expect a symbol and return a callback function:

```
(define-type Posn
  (case-> (-> 'x (-> Real))
          (-> 'y (-> Real))))
```

This Posn type cannot be converted to a contract because there are two cases with the same input arity. Thus zombie uses an alternative type and changes its implementation to match:

```
(define-type Posn
  (-> Symbol
      (U (Pairof 'x (-> Real))
         (Pairof 'y (-> Real)))))
```

Fourth, polymorphic data structures cannot cross Typed Racket boundaries. A benchmark must use monomorphic instantiations of structs, arrays, etc. instead.

Lastly, struct types require particular care at boundaries. In most cases, benchmarks must create an *adaptor module* for each module that exports a struct.

## 4.1 Adaptor Modules

Adaptor modules solve a problem with gradual typing and generative types. In short, they provide a common type definition for several clients to reuse. Creating an adaptor calls for a reorganization of the module dependencies in a benchmark, but it is necessary to allow generative types across a boundary while keeping only two versions of each migratable module. Many GTP Benchmarks use adaptors (14 of 21, table 1).

The most common source of generative types in Racket are struct declarations. Definining a struct `point` with fields `x` and `y` makes a fresh datatype that is incompatible with other structs—even if they have the same name and fields. Typed Racket struct types are generative in the same way; each definition is unique and incompatible with others. This behavior is problematic when two typed modules attempt to import the same untyped struct, because the types generated by each import are incompatible. The solution is to create a typed module that makes a canonical struct definition and exports it to typed code (fig. 4).

One important subtlety arises when creating a lattice of configurations from typed code with adaptors: the adaptor should make a generative type only in configurations where the original struct is untyped. If the original is typed, it should pass through the adaptor to clients. The GTP Benchmarks use an external package to handle the pass-through behavior [124].

## 4.2 Why Two Copies?

The GTP Benchmarks ship two copies of each program: untyped and fully typed. In principle, only the typed copy should be necessary. Practical concerns have kept the untyped code around:

- Having the untyped code on hand makes it easy to compare the benchmark to the program from which it originated, supporting reproducibility studies of the benchmarks' development.
- Any casts in the typed code must be mirrored by equivalent checks in the untyped code. In Typed Racket, casts come from the `cast`, `define-predicate`, and `assert` forms. The first two use types directly, and must be rewritten or factored out to a helper module. An `assert` is simple to translate when it uses built-in predicate functions, but when it uses bespoke predicates the untyped code needs a matching predicate definition. In this case, the untyped code is more than a type-free version of the typed code.
- Typed and untyped code may need to import different modules. Typed code often uses `require-typed-check` [124] to optionally installs boundary checks. Untyped code does not need this dependency.

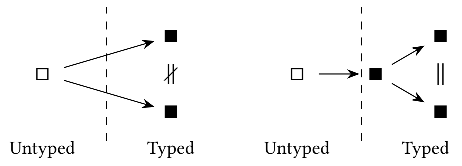


**Figure 4: Adaptor modules provide a canonical definition for generative types**

## 5 SOFTWARE FOR REPRODUCIBILITY

To encourage rigorous, reproducible experiments, the GTP Benchmarks pair with software for: installing the benchmarks (section 5.1), running a full experiment (section 5.2), and plotting the results (section 5.3). An auxilliary package runs quick experiments using representative configurations (rather than a full lattice) for continuous performance monitoring (section 5.4).

## 5.1 Building Configurations

A gradual typing benchmark is a family of configurations related to one program (section 2.2). In the GTP suite, there are over 40,000 configurations in total. While one could imagine working with these configurations directly, it is much more efficient to generate them systematically.

The GTP Benchmarks therefore ship with with minimal source code and scripts for generating configurations. Installing the package furthermore installs necessary dependencies, such as a Typed Racket library for module imports [124], and builds documentation, which presents an overview similar to section 3. In fact, all the data presented in section 3 was originally computed by scripts from the benchmark package—including a simple version of the fsmoo module graph (fig. 3).

On a lower level, each benchmark has a main module (called `main.rkt`) that initializes a computation. The current defaults are the result of experiments to find a large-enough problem that does not run overwhelmingly slowly in the worst case. Still, it may be useful to try other inputs. Some benchmarks, such as suffixtree, have alternative input files available. Others, such as dungeon, have parameters that can be increased or decreased.

```
#lang gtp-measure/manifest
#:config #hash(
  (bin . "/home/gtp/racket-8.8/bin/")
  (cutoff . 6) (num-samples . 10))
("/home/gtp/benchmarks/morsecode" . typed-untyped)
("/home/gtp/benchmarks/take5"     . typed-untyped)
```

**(a) Experiment specification**

```
#lang gtp-measure/output/typed-untyped
("0000" ("cpu time: 602 real time: ..."))
("0010" ("cpu time: 191172 real time: ..."))
...
```

**(b) Output for a configuration lattice**

**Figure 5: Two DSLs for reproducible experiments**

## 5.2 Running an Experiment

Measuring the performance of all configurations for several benchmarks is a nontrivial task. Each configuration needs to run multiple times to deal with uncertainty in the measurements [98, 41] If the goal is to sample instead of running all configurations [58], then samples must be chosen in advance. For experiments running on a compute cluster, where lost connections and timeouts can stop a job, checkpointing and resumption are critical.

The gtp-measure package is a toolkit for reproducible experiments [122]. Given a declaration of programs to run (these programs need not be GTP Benchmarks), it prepares a checklist of configurations, assembles and runs configurations one-by-one (to minimize space costs), and records output in a structured format. When interrupted, an experiment can be resumed from the checklist and current output. As it proceeds, the tool logs information about progress and errors.

In keeping with the DSL-oriented Racket tradition [37], the toolkit comes with little languages to suit the needs of practitioners. One *manifest language* is for preparing experiments. Three *data languages* (with potentially more to come in the future) document the output data. Figure 5 sketches two examples: a manifest that exhaustively measures small benchmarks (with fewer than six modules) and samples large ones by running ten little experiments (fig. 5a), and a data file for one benchmark (fig. 5b). Running the data file (`racket file.rkt`) prints summary statistics, including the number of configurations and the worst-case running time.

## 5.3 Visualizing Results

Finding ways to visualize an exponentially-large dataset was a major challenge for early work on gradual typing performance [131]. Lattice-based (fig. 2) visualizations do not scale, nor do they make it easy to compare two datasets. A better method is to focus on overhead and count the number of configurations that run within some tolerance bound, say, 2x for testing.

Overhead plots, such as the one in fig. 6, count configurations for a range of bounds. In detail, the $x$-axis sets bounds between 1x and 20x; the $y$-axis counts configurations from 0 % to 100 %; and the curve shows how the percent of fast-enough configurations increases as the upper bound increases. The takeaway from fig. 6 is that half the configurations run quickly (within 2x) and the other half is very slow (worse than 20x).

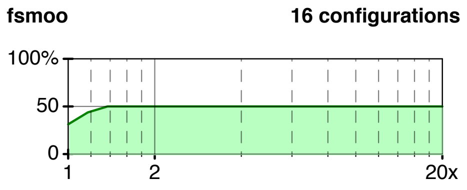


**Figure 6: Overhead plot for fsmoo**

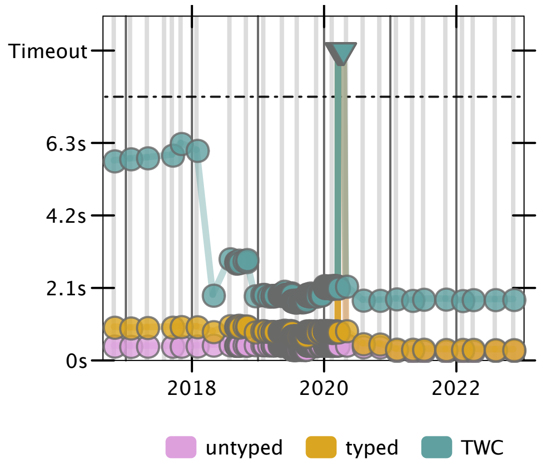


**Figure 7: Monitoring three acquire configurations**

When two curves appear in the same plot, the one with a larger area under it corresponds to better performance. Prior work has many examples, e.g., [57, 10].

The gtp-plot package [123] makes it easy to create overhead plots from experiment data (fig. 5b). For customizing the results, it has parameters to change the colors, font size, legends, and so on. In addition to overhead plots, it supports other visualizations that have been appeared in the literature [10, 58]. More visualizations are on the way [40].

## 5.4 Performance Monitoring

The gtp-checkup package contains a small experiment derived from the benchmarks suitable for running as a nightly task. Instead of measuring all configurations or even a random sample, which would require days of work, it uses three extreme configurations: the untyped, typed, and TWC (typed worst case, section 3.2).

Figure 7 plots results for one benchmark, acquire, across 4+ years of Racket development. Each thin vertical line corresponds to a Racket release. This benchmark sped up considerably in 2018, experienced a brief regression in 2020, and then returned to normal. Similar plots for the other benchmarks appear in the gtp-checkup documentation [29].

## 6 LESSONS LEARNED

Creating a benchmark suite that enables reproducibility is a tall order, especially for an emerging area. On one hand it is important to focus on a specific problem, but on the other hand the significant problems may be unclear and may shift with changes in implementation technology. Even with a problem in hand, it may be challenging to develop evaluation criteria. Our experience with the GTP Benchmarks motivates the following hints to benchmark designers, which hopefully reduce the risk in future efforts:

*Favor broad research questions.* Start with research questions that are broadly applicable and come with few assumptions. Later on, consider narrowing the questions. The GTP Benchmarks make no assumptions about which modules will be equipped with types, thus the results may be overly conservative but apply to all sorts of users. Finding common modes of use is an important topic for user studies, but was not a critical bottleneck for research on gradual typing performance.

*Let end-users drive evaluation criteria.* To design evaluation criteria for a system, put yourself in the shoes of someone using the system who has limited time to run experiments. When developing the GTP Benchmarks, we were tempted to study worst-case overhead, average overhead, and statistical tests about the distribution of fast configurations. Answers to these questions would be fascinating, but useless to working programmers who try gradual typing on one configuration and give up when it runs slowly. Instead, we focused on counting fast configurations to assess overall feasibility. This narrow focus was key to interpreting the large datasets and to the development of a linear sampling method to approximate the outcome for huge datasets [59, 58].

*Keep the benchmark codebase simple.* Benchmarks must be easy to share and run. Make sure they are easy to install, have few dependencies, and have a clear structure (microservice style [138]). The pre-release GTP Benchmarks [132] came with execution scripts, analysis scripts, and code for a paper based on the analysis. Similarly, our Reticulated Python benchmarks come with scripts and a paper [58]. This tight coupling inhibits reuse.

*Use CloudLab.* CloudLab is a tremendous resource for repeatable experiments [30]. Getting started with Cloudlab was easier than any other cluster we have used, and its lack of persistent storage encourages good DevOps practices [159, 70]. Today, we prefer using CloudLab over local desktop machines and have published profiles for running the benchmarks[2] and running a checkup.[3]

## 7 RELATED WORK

Inspiration for the benchmarks came from two main sources: prior benchmark suites, notably DaCapo for Java [12] and the Gabriel Lisp Benchmarks [39]; and Takikawa's *all configurations* experimental method for gradual typing performance [131, 133]. Several works provided helpful guidance. Vitek and Kalibera [154] lay out principles for empirical evaluation. Mytkowicz et al. [98] report pitfalls to avoid when measuring performance. Georges et al. [41] propose a method for rigorous performance evaluation of a managed programming languages. Kistowski et al. [75] list characteristics of a standard benchmark.

Within gradual typing, other performance benchmarks exist for Reticulated Python [58, 157], Grift [76], and Grace [108]. Though smaller in scale, these benchmarks address the measurement question for fine-grained gradual typing in which any *variable* can be typed or untyped. Other gradual benchmark suites focus on type migration [19, 91, 102] and debugging [18, 79].

Today, there are many resources available to benchmark suite curators. SIGSOFT [119] and SIGPLAN [118] offer guidelines for benchmarks, and for empirical research in general. Tools such as ReBench [87, 88] and Krun [8] are free to use and learn from (both influenced our work). The Popper Convention shows the benefits of managing experiments as software projects [74, 73, 72]. Docker is another path to reproducible research [13].

## 8 DISCUSSION

Reproducible experiments are critical for sound gradual typing. The idea of incrementally adding sound types to a large codebase is compelling, and may indeed change the future of programming [21] if researchers develop techniques to eliminate or avoid the cost of soundness in practical situations. Thorough, repeated evaluation is the way forward to identify effective techniques.

The GTP Benchmark Suite combines relevant programs, rigorous experimental methods, and tools for reproducible experiments. It consists of 21 program families that jointly explore 43,972 ways (dominated by the largest families) of mixing typed and untyped code. The programs vary in size, purpose, and method to stress all sorts of type boundaries. Some make heavy use of first-class classes, others use higher-order functions, and still others focus on simple but large data structures. The reproducibility tools prepare gradual configurations, configure interruptible experiments, and visualize results. Much more than icing on top, these tools provide fundamental infrastructure that encodes *why* and *how* to apply the benchmarks to important performance questions.

Two important topics for future work are to develop a language-independent benchmark specification and to develop software metrics tailored to gradual typing. These topics go hand in hand. A specification of the essential language features in each benchmark, similar to FPBench [24] or B2T2 [83], would improve the state of cross-language reproducibility. Currently, ports of the GTP Benchmarks to other languages happen in an ad-hoc way [76, 107, 10, 96, 58, 156]. Software metrics are a tool for quantifying features. Standard metrics exist, e.g., for object-oriented code [22]; gradual typing would benefit from metrics for boundaries, types, and wrappers [65, 47, 38, 114]. Feature-specific profiling [6] may provide a starting point for metrics based on dynamic analysis.

*Data Availability Statement.* The GTP Benchmarks and companion software are available on Software Heritage [121, 122, 123] and Zenodo [49].

## ACKNOWLEDGMENTS

Thanks to Asumu Takikawa, Dan Feltey, Max S. New, Jan Vitek and Matthias Felleisen for collaborating on the initial benchmark suite. Thanks to Sam Tobin-Hochstadt, Spenser Bauman, and Robert

[2]https://www.cloudlab.us/p/rational-prog/gtp-benchmarks, accessed 2023-05-31
[3]https://www.cloudlab.us/p/rational-prog/gtp-checkup, accessed 2023-05-31

**Table 5: Procedure wrapper info**

| Benchmark | Proc apps | Proc makes | Proc depth |
|---|---|---|---|
| sieve | 5 | 20 | 0 |
| forth | 152,149 | 121,788 | 4 |
| fsm | 1,010 | 1,222 | 0 |
| fsmoo | 2,300,769 | 12,578,624 | 2 |
| mbta | 498,809 | 67,997 | 2 |
| morsecode | 3 | 15 | 0 |
| zombie | 536,110 | 689,726 | 13 |
| zordoz | 572,191 | 644,169 | 2 |
| dungeon | 723,105 | 6,899,677 | 2 |
| jpeg | 24 | 258 | 0 |
| lnm | 1,908 | 3,630 | 2 |
| suffixtree | 3,935,912 | 194,764 | 2 |
| kcfa | 3,584 | 1,473 | 0 |
| snake | 11,739,420 | 231 | 0 |
| take5 | 2,717,007 | 9,316,049 | 2 |
| tetris | 5,071 | 252 | 0 |
| acquire | 2,098,839 | 5,350,503 | 15 |
| synth | 30,332,600 | 848 | 3 |
| gregor | 281 | 318 | 2 |
| quadT | 72,589 | 5,640 | 3 |
| quadU | 73,949 | 5,634 | 3 |

**Table 6: Struct wrapper info**

| Benchmark | Struct apps | Struct makes | Struct depth |
|---|---|---|---|
| sieve | 10 | 20 | 2 |
| forth | 51,801,426 | 4,599,325 | 3 |
| fsm | 2,524 | 48 | 2 |
| fsmoo | 28,285,389 | 1,473,316 | 3 |
| mbta | 949,909 | 55 | 2 |
| morsecode | 10 | 20 | 2 |
| zombie | 10 | 20 | 2 |
| zordoz | 317,828 | 8,586 | 2 |
| dungeon | 15,789,250 | 931,444 | 2 |
| jpeg | 25 | 46 | 2 |
| lnm | 30,703 | 396 | 2 |
| suffixtree | 10 | 20 | 2 |
| kcfa | 10 | 20 | 2 |
| snake | 22 | 44 | 2 |
| take5 | 25,940,019 | 1,198,020 | 3 |
| tetris | 22 | 44 | 2 |
| acquire | 8,634,064 | 243,456 | 3 |
| synth | 352 | 20 | 2 |
| gregor | 207 | 53 | 2 |
| quadT | 116,335 | 88,820 | 184 |
| quadU | 115,996 | 88,574 | 184 |

Bruce Findler for improvements to Racket, Typed Racket, and the benchmarks in response to the initial results. Findler suggested the performance monitoring package (section 5.4). Thanks to John Clements, Michael Ballantyne, Matthew Butterick, Matt Might, Danny Yoo, Neil Toronto, John Zeppieri, David Van Horn, Vincent St-Amour, Linh Chi Nguyen, Dee Glaze, and Neil Van Dyke for writing the code that the GTP Benchmarks are based on. Thanks to Lukas Lazarek, Zeina Migeed, Cameron Moy, and Samanvitha Sundar for bug reports and contributions to the benchmarks. Thanks to Shriram Krishnamurthi and the REP'23 reviewers for raising the level of discourse in the paper. Special thanks to Konstantinos Athanasiou for teaching us that "GTP" is Greek internet slang for poor performance (Example: *How are the latest benchmark results? GTP*). This work was supported by NSF grant 2030859 to the CRA for the CIFellows project.

## A WRAPPERS: APPLICATIONS, CONSTRUCTIONS, AND MAX DEPTH

Tables 5 to 7 report low-level details about wrappers. There are three kinds of wrappers in the tables—for procedures (Proc), structs (Struct), and array (Arr)—and three statistics for each: the number of times a wrappers is used or read from (apps), the number of times a wrappers is initialized (makes), and the maximum number of layered wrappers around any value (depth).

**Table 7: Array wrapper info**

| Benchmark | Array apps | Array makes | Array depth |
|---|---|---|---|
| sieve | 0 | 0 | 0 |
| forth | 0 | 0 | 0 |
| fsm | 3,305,323 | 5,006 | 13 |
| fsmoo | 17,952,160 | 9,431,184 | 3 |
| mbta | 0 | 0 | 0 |
| morsecode | 0 | 0 | 0 |
| zombie | 0 | 0 | 0 |
| zordoz | 88 | 67 | 0 |
| dungeon | 1,221,200 | 1,008,400 | 2 |
| jpeg | 5,899,946 | 1,582,130 | 3 |
| lnm | 0 | 3 | 0 |
| suffixtree | 108,592 | 48,672 | 0 |
| kcfa | 0 | 0 | 0 |
| snake | 0 | 0 | 0 |
| take5 | 0 | 0 | 0 |
| tetris | 0 | 0 | 0 |
| acquire | 0 | 0 | 0 |
| synth | 43,593,686 | 43,557,896 | 13 |
| gregor | 116,980 | 29,009 | 2 |
| quadT | 125,168 | 2,788 | 13 |
| quadU | 124,734 | 2,785 | 13 |

## REFERENCES

[1] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. *PACMPL*, 1, ICFP, 39:1–39:28. DOI: 10.1145/3110283.

[2] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2013. Gradual typing for Smalltalk. *SCP*, 96, 1, 52–69.

[3] Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. 2014. Confined gradual typing. In *OOPSLA*, 251–270. DOI: 10.1145/2660193.2660222.

[4] Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. 2020. Reconciling noninterference and gradual typing. In *LICS*. ACM, 116–129. DOI: 10.1145/3373718.3394778.

[5] Vincent St-Amour. [n. d.] Software: synth. Retrieved Feb. 20, 2023 from http://github.com/stamourv/synth.

[6] Leif Andersen, Vincent St-Amour, Jan Vitek, and Matthias Felleisen. 2019. Feature-specific profiling. *TOPLAS*, 41, 1, 4:1–4:34. DOI: 10.1145/3275519.

[7] John Backus. 1978. The history of Fortran I, II, and III. *ACM Sigplan Notices*, 13, 8, 165–180.

[8] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *PACMPL*, 1, OOPSLA, 52:1–52:27. DOI: 10.1145/3133876.

[9] Victor R. Basili. 1996. The role of experimentation in software engineering: past, current, and future. In *ICSE*. IEEE Computer Society, 442–449. Retrieved May 23, 2023 from http://portal.acm.org/citation.cfm?id=227726.227818.

[10] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound gradual typing: only mostly dead. *PACMPL*, 1, OOPSLA, 54:1–54:24. DOI: 10.1145/3133878.

[11] Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*, 257–281. DOI: 10.1007/978-3-662-44202-9_11.

[12] S. M. Blackburn et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 169–190.

[13] Carl Boettiger. 2015. An introduction to docker for reproducible research. *SIGOPS Operating Systems Research*, 49, 1, 71–79. DOI: 10.1145/2723872.2723882.

[14] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: pypy's tracing JIT compiler. In *ICOOOLPS*. ACM, 18–25. DOI: 10.1145/1565824.1565827.

[15] David Broman and Jeremy G. Siek. 2018. Gradually typed symbolic expressions. In *PEPM*. ACM. DOI: 10.1145/3162068.

[16] Matthew Butterick. [n. d.] Software: `quadT`. Retrieved Feb. 20, 2023 from https://github.com/mbutterick/quad/tree/no-check.

[17] Matthew Butterick. [n. d.] Software: `quadU`. Retrieved Feb. 20, 2023 from https://github.com/mbutterick/quad/tree/master.

[18] John Peter Campora III and Sheng Chen. 2020. Taming type annotations in gradual typing. *PACMPL*, 4, OOPSLA, 191:1–191:30. DOI: 10.1145/3428259.

[19] John Peter Campora III, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating gradual types. *PACMPL*, 2, POPL, 15:1–15:29. DOI: 10.1145/3158103.

[20] Guiseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual typing: a new perspective. *PACMPL*, 3, POPL, 16:1–16:32. DOI: 10.1145/3290329.

[21] 2019. *The next 7000 programming languages. Computing and Software Science: State of the Art and Perspectives*, 250–282. ISBN: 978-3-319-91908-9.

[22] Shyam R. Chidamber and Chris F. Kemerer. 1991. Towards a metrics suite for object oriented design. In *OOPSLA*. ACM, 197–211. DOI: 10.1145/117954.117970.

[23] John B. Clements. [n. d.] Software: `morse-code-trainer`. Retrieved Feb. 20, 2023 from https://github.com/jbclements/morse-code-trainer/tree/master/morse-code-trainer.

[24] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. 2016. Toward a standard benchmark format and suite for floating-point analysis, 63–77. DOI: 10.1007/978-3-319-54292-8_6.

[25] [n. d.] DefinitelyTyped: dependency graph. Retrieved Feb. 20, 2023 from https://github.com/DefinitelyTyped/DefinitelyTyped/network/dependents.

[26] R Developers. [n. d.] R: the R project for statistical computing. Retrieved May 26, 2021 from https://www.r-project.org.

[27] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2018. Parametricity versus the universal type. *PACMPL*, 2, POPL, 38:1–38:23. DOI: 10.1145/3158126.

[28] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2022. Two parametricities versus three universal types. *TOPLAS*, 44, 4, 23:1–23:43. DOI: 10.1145/3539657.

[29] [n. d.] Documentation: `gtp-checkup`. Retrieved Feb. 20, 2023 from https://docs.racket-lang.org/gtp-checkup/index.html.

[30] Dmitry Duplyakin et al. 2019. The design and operation of CloudLab. In *USENIX ATC*, 1–14. Retrieved May 23, 2023 from https://www.flux.utah.edu/paper/duplyakin-atc19.

[31] Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional equality for gradual dependently typed programming. *PACMPL*, 6, ICFP, 165–193. DOI: 10.1145/3547627.

[32] Matthias Felleisen. [n. d.] Software: `acquire`. Retrieved Feb. 20, 2023 from http://github.com/mfelleisen/Acquire.

[33] Matthias Felleisen. [n. d.] Software: `sample-fsm`. Retrieved Feb. 20, 2023 from https://github.com/mfelleisen/sample-fsm.

[34] Matthias Felleisen. [n. d.] Software: `sample-fsm/VOO`. Retrieved Feb. 20, 2023 from https://github.com/mfelleisen/sample-fsm/tree/master/VOO.

[35] Matthias Felleisen. [n. d.] Software: `take5`. Retrieved Feb. 20, 2023 from https://github.com/mfelleisen/take5.

[36] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs*. (second ed.). MIT Press. ISBN: 978-0262534802. Retrieved Feb. 1, 2023 from http://www.htdp.org/.

[37] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2015. The racket manifesto. In *SNAPL* (LIPIcs). Vol. 32. Schloss Dagstuhl, 113–128. DOI: 10.4230/LIPIcs.SNAPL.2015.113.

[38] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible contracts: fixing a pathology of gradual typing. *PACMPL*, 2, OOPSLA, 133:1–133:27. DOI: 10.1145/3276503.

[39] Richard P. Gabriel. 1985. *Performance and Evaluation of LISP Systems*. (1st ed.). MIT Press. ISBN: 9780262256193.

[40] Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Which of my transient type checks are not (almost) free? In *VMIL*, 58–66. DOI: 10.1145/3358504.3361232.

[41] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *OOPSLA*. ACM, 57–76. DOI: 10.1145/1297027.1297033.

[42] [n. d.] Ghc: the glasgow haskell compiler. Retrieved Feb. 20, 2023 from https://www.haskell.org/ghc/.

[43] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification*. Addison-Wesley Professional.

[44] Paul Graham. [n. d.] Beating the averages. Retrieved Feb. 20, 2023 from http://www.paulgraham.com/avg.html.

[45] Jim Gray, (Ed.) 1993. *The Benchmark Handbook for Database and Transaction Systems*. (2nd ed.). Morgan Kaufmann. ISBN: 1-55860-292-5.

[46] Michael Greenberg. 2016. Space-efficient latent contracts. In *TFP*. Vol. 10447. Springer, 3–23. DOI: 10.1007/978-3-030-14805-8_1.

[47] Michael Greenberg. 2015. Space-efficient manifest contracts. In *POPL*, 181–194.

[48] Michael Greenberg. 2019. The dynamic practice and static theory of gradual typing. In *SNAPL*, 6:1–6:20. DOI: 10.4230/LIPIcs.SNAPL.2019.6.

[49] [SW] Ben Greenman, Artifact: GTP Benchmarks for Gradual Typing Performance version latest, June 2023. DOI: 10.5281/zenodo.7996759.

[50] Ben Greenman. 2022. Deep and shallow types for gradual languages. In *PLDI*, 580–593. DOI: 10.1145/3519939.3523430.

[51] Ben Greenman. [n. d.] Software: `forth`. Retrieved Feb. 20, 2023 from https://github.com/bennn/forth.

[52] Ben Greenman. [n. d.] Software: `lnm`. Retrieved Feb. 20, 2023 from https://github.com/nuprl/gradual-typing-performance/tree/master/paper/popl-2016/scripts.

[53] Ben Greenman. [n. d.] Software: `zordoz`. Retrieved Feb. 20, 2023 from http://github.com/bennn/zordoz.

[54] Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. Typed–untyped interactions: a comparative analysis. *TOPLAS*, 45, 4, 1–54, 1. DOI: 10.1145/3579833.

[55] Ben Greenman and Matthias Felleisen. 2018. A spectrum of type soundness and performance. *PACMPL*, 2, ICFP, 71:1–71:32.

[56] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete monitors for gradual types. *PACMPL*, 3, OOPSLA, 122:1–122:29. DOI: 10.1145/3360548.

[57] Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. 2022. A transient semantics for Typed Racket. *Programming*, 6, 2, 9:1–9:26. DOI: 10.22152/programming-journal.org/2022/6/9.

[58] Ben Greenman and Zeina Migeed. 2018. On the cost of type-tag soundness. In *PEPM*, 30–39. DOI: 10.1145/3162066.

[59] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to evaluate the performance of gradual type systems. *JFP*, 29, e4, 1–45. DOI: 10.1145/3473573.

[60] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: hybrid checking for flexible specifications. In *SFP. University of Chicago, TR-2006-06*, 93–104. http://scheme2006.cs.uchicago.edu/scheme2006.pdf.

[61] [n. d.] GTP Benchmarks documentation. Retrieved Sept. 9, 2022 from https://docs.racket-lang.org/gtp-benchmarks/.

[62] [n. d.] GTP Benchmarks source code. Retrieved Sept. 6, 2022 from https://github.com/utahplt/gtp-benchmarks.

[63] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The mälardalen WCET benchmarks: past, present and future. In *WCET* (OASIcs). Vol. 15. Schloss Dagstuhl, 136–146. DOI: 10.4230/OASIcs.WCET.2010.136.

[64] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: a ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4, 3, 1–29.

[65] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *HOSC*, 23, 2, 167–189. DOI: 10.1007/s10990-011-9066-z.

[66] Rich Hickey. [n. d.] Effective programs: 10 years of clojure. Retrieved Feb. 20, 2023 from https://youtu.be/2V1FtfBDsLU.

[67] Charles A Hoare. 1973. Hints on programming language design. Tech. rep. Stanford University.

[68] Benjamin D. Horne, Maurício Gruppi, Kenneth Joseph, Jon Green, John P. Wihbey, and Sibel Adali. 2022. NELA-Local: A dataset of U.S. local news articles for the study of county-level news ecosystems. In *ICWSM*. AAAI Press, 1275–1284. Retrieved May 25, 2023 from https://ojs.aaai.org/index.php/ICWSM/article/view/19379.

[69] Hansen Hsu. [n. d.] Oral history of Guido Van Rossum. Retrieved Feb. 20, 2023 from https://archive.computerhistory.org/resources/access/text/2018/07/102738719-05-01-acc.pdf.

[70] Michael Httermann. 2012. *DevOps for developers*. (1st ed.). Apress. ISBN: 978-1-4302-4569-8.

[71] Sujay Jayakar. 2020. Rewriting the heart of our sync engine. Retrieved Feb. 20, 2023 from https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine.

[72] Ivo Jimenez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Jay Lofstead, Carlos Maltzahn, Kathryn Mohror, and Robert Ricci. 2017. PopperCI: automated reproducibility validation. In *INFOCOM WKSHPS*, 450–455. DOI: 10.1109/INFCOMW.2017.8116418.

[73] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay F. Lofstead, Kathryn Mohror, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2016. Standing on the shoulders of giants by managing scientific experiments like software. *Usenix; login*, 41, 4. Retrieved May 23, 2023 from https://www.usenix.org/publications/login/winter2016/jimenez.

[74] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2017. The Popper convention: making reproducible systems evaluation practical. In *IPDPSW*. IEEE, 1561–1570. DOI: 10.1109/IPDPSW.2017.157.

[75] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to build a benchmark. In *ICPE*. ACM, 333–336. ISBN: 9781450332484. DOI: 10.1145/2668930.2688819.

[76] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward efficient gradual typing for structural types via coercions. In *PLDI*, 517–532. DOI: 10.1145/3314221.3314627.

[77] Elizabeth Labrada, Matías Toro, Éric Tanter, and Dominique Devriese. 2022. Plausible sealing for gradual parametricity. *PACMPL*, 6, OOPSLA1, 1–28. DOI: 10.1145/3527314.

[78] Leslie Lamport and Lawrence C. Paulson. 1999. Should your specification language be typed. *TOPLAS*, 21, 3, 502–526. DOI: 10.1145/319301.319317.

[79] Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to evaluate blame for gradual types. *PACMPL*, 5, ICFP, 1–29. DOI: 10.1145/3473573.

[80] Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2023. How to evaluate blame for gradual types, part 2. *PACMPL*, 7, ICFP, (accepted for publication).

[81] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41, 12, 1236–1256.

[82] Erwan Lemonnier. 2006. Pluto: or how to make Perl juggle with billions. Accessed 2020-08-25. (2006). http://erwan.lemonnier.se/talks/pluto.html.

[83] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. 2022. Types for tables: a language design benchmark. *Programming*, 6, 2, 8:1–8:30.

[84] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2023. Gradual soundness: lessons from static python. *Programming*, 7, 1, 2:1–2:40.

[85] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A reasonably gradual type theory. *PACMPL*, 6, ICFP, 931–959. DOI: 10.1145/3547655.

[86] Stefan Malewski, Michael Greenberg, and Éric Tanter. 2021. Gradually structured data. *PACMPL*, 5, OOPSLA, 1–29. DOI: 10.1145/3485503.

[87] Stefan Marr. 2018. Rebench: execute and document benchmarks reproducibly. Version 1.0. (Aug. 2018). DOI: 10.5281/zenodo.1311762.

[88] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language compiler benchmarking: are we fast yet? In *DLS*. ACM, 120–131. DOI: 10.1145/2989225.2989232.

[89] Jacob Matthews and Robert Bruce Findler. 2009. Operational semantics for multi-language programs. *TOPLAS*, 31, 3, 1–44. DOI: 10.1145/1498926.1498930.

[90] John McCarthy, Paul W Abrahams, Daniel J Edwards, Timothy P Hart, and Michael I Levin. 1962. *LISP 1.5 programmer's manual*. MIT press.

[91] Zeina Migeed and Jens Palsberg. 2020. What is decidable about gradual types? *PACMPL*, 4, POPL, 29:1–29:29. DOI: 10.1145/3371097.

[92] Matt Might. [n. d.] *k*-CFA: determining types and/or control-flow in languages like Python, Java and Scheme. Retrieved Feb. 20, 2023 from https://matt.might.net/articles/implementation-of-kcfa-and-0cfa/.

[93] Robin Milner. 1987. Is computing an experimental science? *Journal of Information Technology*, 2, 2, 58–66. DOI: 10.1057/jit.1987.12.

[94] David A. Moon. 1974. MACLISP Reference Manual, Revision 0. Tech. rep. MIT.

[95] Cameron Moy, Phuc C. Nguyen, Sam Tobin–Hochstadt, and David Van Horn. 2021. Corpse reviver: sound and efficient gradual typing via contract verification. *PACMPL*, 5, POPL, 1–28. DOI: 10.1145/3434334.

[96] Fabian Muehlboeck and Ross Tate. 2017. Sound gradual typing is nominally alive and well. *PACMPL*, 1, OOPSLA, 56:1–56:30. DOI: 10.1145/3133880.

[97] Fabian Muehlboeck and Ross Tate. 2021. Transitioning from structural to nominal code with efficient gradual typing. *PACMPL*, 5, OOPSLA, 127:1–127:29. DOI: 10.1145/3485504.

[98] Todd Mytkowicz, Amer Diwan, Matthais Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong. In *ASPLOS*, 265–276.

[99] Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and parametricity: together again for the first time. *PACMPL*, 4, POPL, 46:1–46:32.

[100] Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual type theory. *PACMPL*, 3, POPL, 15:1–15:31. DOI: 10.1145/3290328.

[101] Jeppe Nørregaard, Benjamin D. Horne, and Sibel Adali. 2019. NELA-GT-2018: A large multi-labelled news dataset for the study of misinformation in news articles. In *ICWSM*. AAAI Press, 630–638. Retrieved May 25, 2023 from https://ojs.aaai.org/index.php/ICWSM/article/view/3261.

[102] Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-based gradual type migration. *PACMPL*, 5, OOPSLA, 1–27. DOI: 10.1145/3485488.

[103] PLT. 2017. The Racket Programming Lanugage. http://racket-lang.org. (2017).

[104] Terence W Pratt. 1987. Programmming languages: design and implementation. (1987).

[105] Aleksandar Prokopec et al. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *PLDI*. ACM, 31–47. DOI: 10.1145/3314221.3314637.

[106] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & efficient gradual typing for TypeScript. In *POPL*, 167–180. DOI: 10.1145/2676726.2676971.

[107] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The vm already knew that: leveraging compile-time knowledge to optimize gradual typing. *PACMPL*, 1, OOPSLA, 55:1–55:27. DOI: 10.1145/3133879.

[108] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient typechecks are (almost) free. In *ECOOP*, 15:1–15:29. DOI: 10.4230/LIPIcs.ECOOP.2019.5.

[109] [n. d.] Rust programming language. Retrieved Feb. 20, 2023 from https://www.rust-lang.org/.

[110] Jean E Sammet. 1985. Brief summary of the early history of cobol. *Annals of the History of Computing*, 7, 4, 288–303.

[111] Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. 2021. Abstracting gradual typing moving forward: precise and space-efficient. *PACMPL*, 5, POPL, 1–28. DOI: 10.1145/3434342.

[112] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *OOPSLA*. ACM, 657–676. DOI: 10.1145/2048066.2048118.

[113] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *SFP. University of Chicago, TR-2006-06*, 81–92. http://scheme2006.cs.uchicago.edu/scheme2006.pdf.

[114] Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2021. Blame and coercion: together again for the first time. *JFP*, 31, e20. DOI: 10.1017/S0956796821000101.

[115] Jeremy G. Siek and Sam Tobin-Hochstadt. 2016. The recursive union of some gradual types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Vol. 9600. Springer, 388–410. DOI: 10.1007/978-3-319-30936-1_21.

[116] Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015. Blame and coercion: together again for the first time. In *PLDI*. ACM, 425–435. DOI: 10.1145/2737924.2737968.

[117] Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015. Monotonic references for efficient gradual typing. In *ESOP*. Springer, 432–456. DOI: 10.1007/978-3-662-46669-8_18.

[118] SIGPLAN. [n. d.] Empirical evaluation guidelines. Retrieved May 23, 2023 from https://sigplan.org/Resources/EmpiricalEvaluation.

[119] SIGSOFT. [n. d.] Empirical standards. Retrieved  from.

[120] S.E. Sim, S. Easterbrook, and R.C. Holt. 2003. Using benchmarking to advance research: a challenge to software engineering. In *ICSE*, 74–83. DOI: 10.1109/ICSE.2003.1201189.

[121] [SW Rel.], Software: GTP Benchmarks version 9.2, 2022. VCS: https://github.com/utahplt/gtp-benchmarks, SWHID: ⟨swh:1:dir:38637b36446d7ff772b635b623a78bc9d01c260b⟩.

[122] [SW Rel.], Software: `gtp-measure` 2022. VCS: https://github.com/utahplt/gtp-measure, SWHID: ⟨swh:1:dir:9ef0b809bd17c71bcb30bc9f0037713a116c9495⟩.

[123] [SW Rel.], Software: `gtp-plot` 2022. VCS: https://github.com/utahplt/gtp-plot, SWHID: ⟨swh:1:dir:ef20cf852c1fb32898624c59e08aae40d1da65cf⟩.

[124] [SW Rel.], Software: `require-typed-check` 2022. VCS: https://github.com/bennn/require-typed-check, SWHID: ⟨swh:1:dir:45d3167754665f60df9bbf92cae6800020ea9b20⟩.

[125] [n. d.] SPEC CPU 2017. Retrieved Feb. 20, 2023 from https://spec.org/cpu2017/.

[126] [n. d.] State of js 2020: javascript flavors. Retrieved Feb. 20, 2023 from https://2020.stateofjs.com/en-US/technologies/javascript-flavors/.

[127] Guy L. Steele Jr. 1990. *Common Lisp.* (2nd ed.). Digital Press.

[128] Michael Stevenson. 2018. Having it both ways: Larry Wall, Perl and the technology and culture of the early web. *Internet Histories*, 2, 3-4, 264–280.

[129] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. 2009. Practical variable-arity polymorphism. In *ESOP*, 32–46.

[130] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*, 943–962.

[131] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. 2015. Towards practical gradual typing. In *ECOOP*, 4–27.

[132] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead? In *POPL*, 456–468. DOI: 10.1145/2837614.2837630.

[133] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2015. Position paper: performance evaluation for gradual typing. In *STOP*. ACM.

[134] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. 2013. Constraining delimited control with contracts. In *ESOP*, 229–248.

[135] Dart Team. [n. d.] The Dart type system. Retrieved Jan. 15, 2022 from https://dart.dev/guides/language/type-system.

[136] [SW] The pandas development team, pandas-dev/pandas: Pandas version latest, Feb. 2020. URL: https://doi.org/10.5281/zenodo.3509134.

[137] [n. d.] The Coq proof assistant. Retrieved June 1, 2023 from https://coq.inria.fr.

[138] Johannes Thönes. 2015. Microservices. *IEEE Software*, 32, 1, 116–116. DOI: 10.1109/MS.2015.11.

[139] Walter F. Tichy. 2014. Where's the science in software engineering? *Ubiquity*, 2014, March. DOI: 10.1145/2590528.2590529.

[140] Walter F. Tichy, Mathias Landhäußer, and Sven J. Körner. 2015. nlrpBENCH: a benchmark for natural language requirements processing. In *Software Engineering & Management.* GI, 159–164. Retrieved May 23, 2023 from https://dl.gi.de/20.500.12116/2542.

[141] W.F. Tichy. 1998. Should computer scientists experiment more? *Computer*, 31, 5, 32–40. DOI: 10.1109/2.675631.

[142] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *DLS*, 964–974. DOI: 10.1145/1176617.1176755.

[143] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. In *POPL*, 395–406.

[144] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory typing: ten years later. In *SNAPL*, 17:1–17:17. DOI: 10.4230/LIPIcs.SNAPL.2017.17.

[145] Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual parametricity, revisited. *PACMPL*, 3, POPL, 17:1–17:30. DOI: 10.1145/3290330.

[146] [n. d.] TypeScript. Retrieved May 18, 2021 from https://www.typescriptlang.org.

[147] [n. d.] V8 Benchmarks. Retrieved May 24, 2023 from https://v8.dev/docs/benchmarks.

[148] Neil Van Dyke. [n. d.] Software: `levenshtein`. Retrieved Feb. 20, 2023 from https://www.neilvandyke.org/racket/levenshtein/.

[149] David Van Horn. [n. d.] Software: `snake`. Retrieved Feb. 20, 2023 from https://github.com/philnguyen/soft-contract/tree/master/soft-contract/benchmark-contract-overhead.

[150] David Van Horn. [n. d.] Software: `tetris`. Retrieved Feb. 20, 2023 from https://github.com/philnguyen/soft-contract/tree/master/soft-contract/benchmark-contract-overhead.

[151] David Van Horn. [n. d.] Software: `zombie`. Retrieved Feb. 20, 2023 from https://github.com/philnguyen/soft-contract/tree/master/soft-contract/benchmark-contract-overhead.

[152] Bill Venners. 2009. Twitter on Scala. Retrieved Feb. 20, 2023 from https://www.artima.com/articles/twitter-on-scala.

[153] Bill Venners and Frank Sommers. [n. d.] Strong versus weak typing: a conversation with guido van rossum, part v. Retrieved Feb. 20, 2023 from https://www.artima.com/articles/strong-versus-weak-typing.

[154] Jan Vitek and Tomas Kalibera. 2011. Repeatability, reproducibility, and rigor in systems research. In *EMSOFT*. ACM, 33–38. DOI: 10.1145/2038642.2038650.

[155] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *DLS*, 45–56.

[156] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and evaluating transient gradual typing. In *DLS*, 28–41. DOI: 10.1145/3359619.3359742.

[157] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *POPL*, 762–774. DOI: 10.1145/3009837.3009849.

[158] Philip Wadler. 1989. Theorems for free! In *FPCA*. ACM, 347–359. DOI: 10.1145/99370.99404.

[159] Adam Wiggins. [n. d.] The twelve-factor app. Retrieved May 23, 2022 from https://12factor.net.

[160] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed messages: measuring conformance and non-interference in TypeScript. In *ECOOP*, 28:1–28:29.

[161] Andy Wingo. [n. d.] Software: `jpeg`. Retrieved Feb. 20, 2023 from https://github.com/wingo/racket-jpeg.

[162] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating typed and untyped code in a scripting language. In *POPL*, 377–388. DOI: 10.1145/1706299.1706343.

[163] Danny Yoo. [n. d.] Software: `suffixtree`. Retrieved Feb. 20, 2023 from https://github.com/dyoo/suffixtree.

[164] Jon Zeppieri. [n. d.] Software: `gregor`. Retrieved Feb. 20, 2023 from https://github.com/97jaz/gregor.