Sound Gradual Typing Performance: Evaluation of Safe TypeScript

Temur Saidkhodjaev
University of Maryland
temurson@umd.edu

William Cao
University of Maryland
wcao12@umd.edu

John Cole
University of Maryland
jackcole@umd.edu

ABSTRACT

Gradual typing offers a compromise between static typing and dynamic typing that is hard to pass up. What's not to love about a language that combines the flexibility of Python with the type-checking of Java? Possibly the performance overhead. In this paper, we sought to evaluate the performance of a sound gradually typed implementation of TypeScript known as Safe TypeScript. In our results, we found that the future of gradual typing may not be as dismal as initially forecast by previous works. In fact, we found that adding more types to a program significantly reduces its runtime compared to a completely untyped benchmark.

Keywords

Gradual Typing; Performance Evaluation; TypeScript

1. INTRODUCTION

Programmers indoctrinated into the dynamically typed or statically typed camps argue relentlessly about which approach is better. Those who prefer dynamic typing embrace flexibility and quick prototyping, while those who like their programs statically typed argue for better error detection, faster runtimes and better software design. Amidst this battle a compromise called gradual typing has been proposed in 2006 [6]. When developing type systems, researchers always try to maintain the soundness of the system, the famous "well-typed programs don't go wrong", and gradual typing is no different. However, due to the nature of gradual typing, the compiler cannot possibly know all the types before runtime, so the solution is to inject runtime type checks to maintain the type safety guarantees. Unfortunately, it seems that this solution can significantly degrade performance of gradually typed programs, so Takikawa et. al., after discovering up to 100x slowdowns in performance of gradual typing implementation for Racket called Typed Racket, have declared sound gradual typing dead [3]. Numerous papers have been published since then: some describing the ways to optimize gradual typing, some evaluating performance in other languages, and it seems that gradual typing is still breathing.

Gradual typing has been implemented in several languages since the day it has been introduced, including the crowd favorite TypeScript. However, gradual typing in TypeScript is by design not sound, meaning that it only checks types at compile time, and provides no guarantees about the runtime type safety. That is why testing TypeScript's performance in the context of evaluating sound gradual typing is meaningless. Nevertheless, there is not one, but two implementations of sound gradual typing for TypeScript, namely StrongScript [5] and Safe TypeScript [2], and the latter even attempts to utilize the type information to increase performance. The authors of these implementations have measured their performance, and found it more than satisfactory, but they have not done the same thorough evaluation as Siek and Taha did for Typed Racket.

This paper carries out the performance evaluation of Safe TypeScript using the methodology by Takikawa et. al. [3] Only the "safe" configuration of the compiler is used to speed up the evaluation process. The benchmark program is RayTracing Engine from Octane.js [4] modified by Rastogi et. al. [2] The resulting data indicates that even though using Safe TypeScript compiler introduces a 10× slowdown with respect to the original TypeScript compiler, adding type annotations to the fully untyped code does not result in pathological slowdowns as in Typed Racket performance evaluation. [3] On the other hand, the more type annotations the programmer adds, the more performance is restored with respect to the original TypeScript compiler. Therefore, we claim that sound gradual typing for TypeScript is definitely alive, yet might need some exercise to get into shape.

The next section of this paper will explore related work on this topic. Then, we discuss the testing framework for this evaluation, the way benchmark programs were rewritten and combined into a set of partially typed programs, our testing setup and tools used to carry out the evaluation. Finally, we present the data and draw a conclusion regarding the vitality of sound gradual typing for TypeScript, and share the insights we gained in the process.

2. BACKGROUND AND RELATED WORK

The goal that we had in mind when conducting our performance evaluations was to compare our results to evaluations of other implementations of gradual typing using the same methods. As a result, the framework that we built our evaluation on bears a striking resemblance to the lattice-style framework described in “Is Sound Gradual Typing Dead?”.

“Is Sound Gradual Typing Dead?” describes the authors’ take on evaluating the performance of Typed Racket, a gradually typed dialect of the Racket programming language. In their evaluation, the performance of Typed Racket was assessed by the runtime of benchmarks using six different modules. Each module had a typed version and an untyped version. The benchmarks were run using every possible permutation of typed and untyped modules, meaning there were $2^6 = 64$ configurations tested. Because the modules were the units of code being changed, the evaluation was of module-level granularity. What the authors discovered was that certain combinations of untyped and typed modules resulted in catastrophic runtime overhead, up to a 105.3x increase over the fully untyped benchmark. As a result, the authors declared “If applying our method to other gradual type implementations yields similar results, then sound gradual typing is dead.” With this statement in mind, we set out to determine if applying their methods to a wholly different implementation of gradual typing would indeed yield similar results.

When we were preparing our benchmarks for evaluation, we based our evaluation method off of Takikawa’s method but made some changes. The most obvious difference is that we were evaluating Safe Typescript instead of Typed Racket, and so we used benchmarks written in Typescript (located in the Github Repository for Safe Typescript). We also changed the level of granularity used for our benchmarks. Rather than using typed and untyped versions of modules, we added types to some classes of the benchmark, ranging from all classes to none of them. Thus *our* level of granularity was the classes of the benchmark. Instead of six groups, we had four, resulting in $2^4 = 16$ different configurations for us to test.

To the best of our knowledge, there have not been any attempts to rigorously evaluate performance of StrongScript [5] and Safe TypeScript [2] gradual typing implementations for TypeScript. This might be due to the fact that TypeScript itself is not sound, and both Safe TypeScript and StrongScript are both rather experimental research efforts, and not widely adopted in the TypeScript community.

For both Safe TypeScript and StrongScript only the performance of fully typed and fully untyped programs has been measured, but intermediate states have not been considered. It is argued that this is unrealistic, since normally programmers tend to refactor their TypeScript code by adding types step-by-step, and it would be impossible to do in one sitting for large programs. Therefore, gradual typing implementers have to make sure that the partially typed program stays as performant as the untyped one.

3. TESTING FRAMEWORK

In this section, we describe our evaluation methodology. Namely, how methodology from “Is Sound Gradual Typing Dead” was adapted for our evaluation, how the code of the benchmark program was divided into several pieces to form the performance lattice, then how the code was modified and the data was obtained.

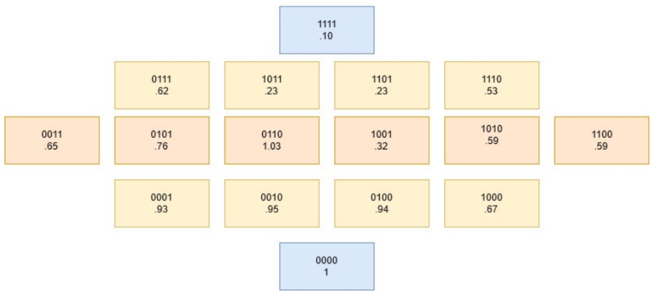
Carrying out the performance evaluation of Safe TypeScript and StrongScript involves several challenges. First, the general framework of testing needs to be decided upon. This has been successfully done by Siek and Taha in their evaluation of Typed Racket, but their approach relies upon the fact that Typed Racket only allows type granularity on the level of modules. In other words, either the entire module is typed, or it is untyped. This is not the case for TypeScript, and since we cannot test all possible combinations of being typed/untyped for every single place types appear in, we have decided to use functions and classes as a unit of program refactoring. Files were not chosen for this, because the benchmark program we use is contained in a single file. The code was divided in logically connected and roughly equally sized pieces (around 200 lines of code each).

We evaluated Safe TypeScript on the benchmark of a Ray-Tracing Engine. The engine has both typed and untyped benchmark. We combine the typed and untyped classes with small amounts of modification to make the code compile. For example, we added return types to functions to smooth out compatibility between partially typed and untyped code. We divide the code into 4 pieces. There was a total of 2 choices for each class as typed or untyped. There were thus 16 different classes that were evaluated.

To select the number of trials we used Benchmark.js library[1]. The library automatically selects the number of trials to run to avoid noise and have enough trials to get a statically significant results. If we were to run it manually we would have potentially unhelpful 0 ms result that are unhelpful for our data collection. In addition once the number of trials that we ran finished we would have to manually determine to have more trails to get a statically significant result. Using this adaptive library that adjusts, the margin of error of each calculated trial was less than .001.

4. RESULTS

We organized the performance data from running our benchmarks into a lattice shape similar to Takikawa et. al. There is a dramatic difference between the results from our performance evaluation of Safe TypeScript and Takikawa et. al's evaluation of Typed Racket. As we can see from the figure below, there is a consistent decrease in runtime as more and more types are added to the benchmark, eventually getting as low as ten percent of the fully untyped runtime. In Takikawa et. al., mixing typed and untyped modules resulted in disastrous runtime increases of up to 105.3× the original fully untyped benchmark [3]. This dramatic difference could possibly be caused by Typed Racket's compiler. If Typed Racket's compiler had any redundant type contract validation, that would certainly contribute to the abysmal runtime observed by Takikawa et. al.



A one represents a typed set of classes, and a zero represents an untyped set. The more typed code that is added to the benchmark, the faster the code runs, up to a ninety percent decrease in runtime.

As we can see, the performance consistently improves with the exception of one aberration where the performance suffers a slight overhead of a 0.03 percent increase in runtime. Overall, however, our results were very satisfactory and we can safely say that sound gradual typing is not dead, at least with regards to the implementation of sound gradual typing for Safe TypeScript.

Our insights are that although we see an improvement in performance in the vacuum of Safe TypeScript, it should be noted that base untyped Safe Typescript results in a 10× performance overhead compared to vanilla TypeScript. As we add more and more types, however, the runtime is reclaimed. Generally speaking, typed code is faster than untyped code. Therefore, fully typing Safe TypeScript results in a program with the same runtime as a regular TypeScript program.

5. CONCLUSION

We can see that for almost all the data, there was a speedup for gradually typed and fully typed variations of the benchmark in all but one case. The speedup in TypeScript was a contrast to what was seen in Takikawa et. al.'s paper. [3]. It is interesting to note that there is a 10× speedup when changing from Safe TypeScript to normal TypeScript, so this indicates that adding types to the program causes the program to gradually regain performance of the code without the runtime checks.

There can be a few explanations for this. First, it has been argued that Typed Racket's compiler is not performing enough optimizations to the typechecking process, thus incurring such a large runtime overhead. It might be just the problem of Typed Racket compiler in this case. Second, we hypothesize that the JavaScript runtime (V8 engine) is performing optimizations so well that the cost of runtime checks becomes negligible.

For future improvements on this project, we can look to evaluating more implementations of sound gradual typing in TypeScript, such as StrongScript, which we mentioned in our introduction. Furthermore, we could improve upon our existing tests by using more Octane benchmarks, as only using the Ray-Tracing Engine benchmark is somewhat limiting. If we so wished, we could also run our benchmarks on different versions of the Safe TypeScript compiler, such as an optimized compiler. Something we wished we could have done but did not do was figure out which typechecking contracts contributed the most to the runtime overhead in the different stages of the lattice. This analysis was done in Takikawa et. al., but we lack the necessary tools to perform it.

Overall, we conclude that sound gradual typing for TypeScript is definitely not dead, yet might need some exercise to get into shape. In particular, modifying the JavaScript runtime to be type-aware and thus safer and faster, or adding some contract optimization strategies are some of the ways to do this.

Acknowledgments

We would like to thank our dedicated teachers Dave Levin and Leilani Battle, who continued to supply us with educational resources and valuable guidance in the midst of a global pandemic.

6. REFERENCES

[1] Mathias Bynens and John-David Dalton. Benchmark JS. https://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/, Dec. 2010.
[2] Aseem Rastogi and Nikhil Swamy and Cedric Fournet and Gavin Bierman and Panagiotis Vekris. Safe Efficient Gradual Typing for TypeScript. http://www.cs.umd.edu/~aseem/safets-tr.pdf, Aug. 2014.
[3] Asumu Takikawa and Daniel Feltey and Ben Greenman and Max S. New and Jan Vitek and Matthias Felleisen. Is Sound Gradual Typing Dead? http://www.ccis.northeastern.edu/home/types/publications/gradual-dead/pre-treatment.pdf, Jan. 2016.
[4] Google Developers. The JavaScript Benchmark Suite for the modern web. https://developers.google.com/octane, Apr. 2017.
[5] Gregor Richards and Francesco Zappa Nardelli and Jan Vitek3. Concrete Types for TypeScript. http://janvitek.org/pubs/ecoop15a.pdf, Apr. 2015.
[6] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. http://schemeworkshop.org/2006/13-siek.pdf, Jan. 2006.