CS842 Type Systems: The Cost of Safety

Jia Wu

Tuesday 25th July, 2017

1 Introduction

Type systems are a tradeoff between security and convenience. Dynamically typed systems are flexible enough to allow rapid implementation and deployment. However, these systems typically have some overhead for runtime checks versus their statically typed counterparts. This paper aims to investigate the performance overhead incurred in a few impartive programming languages via simple benchmarks.

Keywords— Compiler, Static Type System, Gradual Type System, PHP, Hack, Python, Cython, JavaScript, TypeScript

2 Related Work

Benchmarking the performance of gradually typed languages is not a novel idea. Groups such as Takikawa et al. [TFG+16] have built a framework for Typed Racket which examines the performance penalties of incrementally typing modules over multiple configurations. These configurations provide a view of how developers may gradually type their programs at a macro-level. This work suggests that incrementally typing a program has significant penalties incurring slowdowns of up to 105x relative to the untyped program. Takikawa et al. discovered that by fully typing all modules within a program, the resulting program has better performance than its untyped equivalent.

Groups such as Rastogi et al. [RSF+15] have developed extensions to compile safe TypeScript code. By conducting a two-phase compilation process which introduces some runtime type information, the authors are able to enforce type safety on a subset of TypeScript code. Again, the property of type safety comes at a cost of a slowdown factor of approximately 1.15x when porting a TypeScript Compiler.

Wilbers et al. [WLØ09] benchmarked the runtime performance of Cython vs Python by implementing the Trapezoidal rule in a function. They observed that Cython actually speeds up the execution of the function by approximately 30x, and noted that it was due to the compilation of Python to C code which allowed for this speedup.

Gregor et al. [RZNV15] have developed StrongScript which is an evolution of TypeScript that provides safety guarantees as well as performance guarantees. Their implementation of StrongScript provides speedups of 22% in a raytracing benchmark, but also has slowdowns

of 0.86% in their richards benchmark. Additionally, this work claims that blame tracking for errors is a prohibitively expensive feature which requires additional work.

3 Method

3.1 Environment

Benchmarks for the following experiments were executed in an Ubuntu Docker container running ontop of an Intel i7 4790k at 4.4GHz with 16GB RAM. Docker was selected as the execution environment as it provides a consistent runtime environment regardless of deployment [Mer14]. The builtin linux time functionality was chosen to measure the elapsed real time betwen invocation and termination of each benchmark. All relevant code can be found at: `https://github.com/JRWu/spring2017_cs842typesystems_final`. Additionally, the docker image containing the execution environment can be sourced by using *docker pull jwu424/cs842*.

3.2 Languages

Three dynamically typed languages and their gradually typed counterparts were chosen for the benchmark purposes: PHP/HACK, Python/Cython, Javascript/Typescript. Hack is a language developed by Adams et al. at Facebook to address the high CPU performance overhead incurred by native PHP running on an interpreter [AEM+14]. Cython is a Python extension which compiles annotated Python to C and subsequently links it for usage by the Python interpreter [BBC+11]. It is gradually typed in the sense that type annotations are optional, and can be incrementally added to a program. TypeScript is an unsound extension of JavaScript, but it adds an additional layer of security in the form of a static type system [BAT14]. The docker container uses Cython version 0.26, Python 2.7.6, HHVM version 3.18.3, PHP version 5.6.99, and npm version 5.3.0 to run JavaScript EMCAScript 5 (ES5) along with TypeScript compiled to ES5.

3.3 Benchmarks

All three languages have the same test suite of benchmarks implemented. Both Cython and TypeScript were compiled prior to execution, while Hack was executed with the HHVM which includes JIT compilation. Functions were called an arbitrary amount of times such as 1000000 or 10000 but remain consistent between the typed and untyped calls. The reason for this iterative function calling process was to generate numbers large enough for the linux time functionality to detect. Parameters passed to the functions are identical between typed and untyped calls with the only difference being the typehints added to typed calls. Runtime is averaged between 10 calls to the file containing the tests, and was measured using the builtin linux *time* function. All tests were run within the docker container to exclude the time necessary to initiate the container.

3.3.1 Arithmetic

A set of arithmetic operations were implemented as functions. These operations are comprised of the functions addition, subtraction, multiplication, division and iteration. For each of the untyped function variants, arguments are passed into the function untyped, and a loop is applied to apply the primitive operation to each type 10 times.

3.3.2 Strings

Two functions are implemented in this test set, a string concatenation function and the levenshtein edit distance function. The levenshtein function computes the minimal number of deletions, insertions or substitutions required to transform a source string s to a target string t.

3.3.3 IO

A simple file opening function which opens a file, reads all the lines, and assigns them to a variable before closing the file.

3.3.4 Recursion

Three recursive functions: fibonnaci, greatest common divisor (gcd), and recursive addition were written for this test set. Fibonacci computes the sum of the fibonnaci sequence up to a given number passed as a parameter. GCD computes the greatest common divisor between two numbers passed in as parameters. Recursive addition adds two numbers recursively and makes n recursive calls where n is one of the parameters. Fibonnaci makes two calls to itself within its function body while gcd and recursive additions make one.

4 Results/Discussion

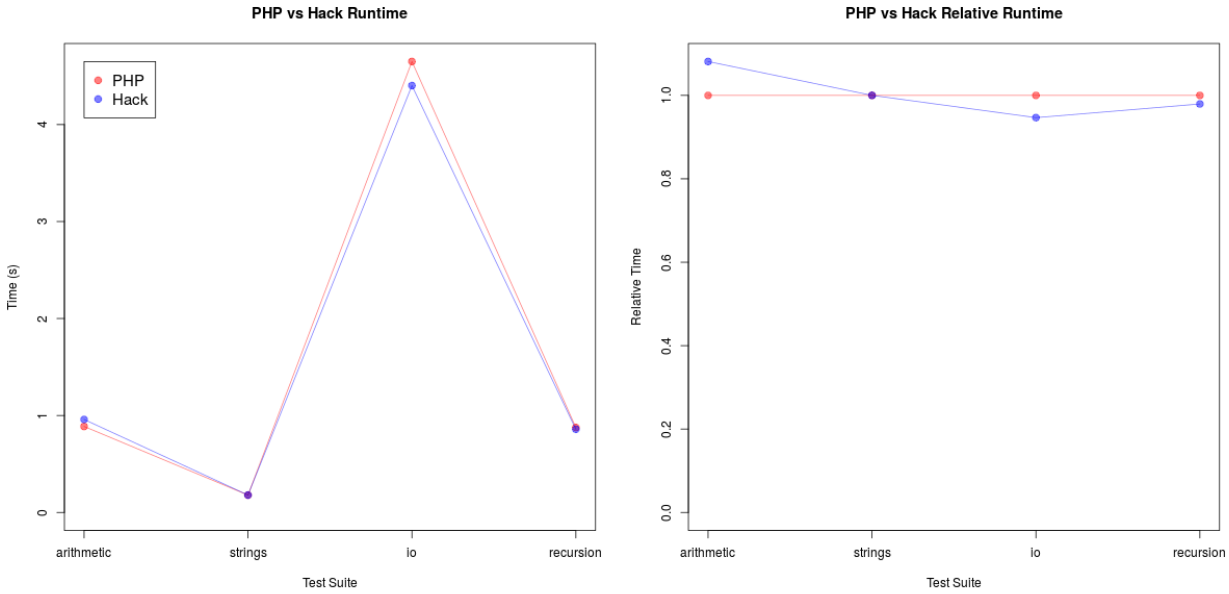
Performance results varied across the board between the three language implementations. Table 1 quantifies the time reduction factor observed in untyped languages relative to their typed counterparts. For example, arithmetic operations in Cython take only 0.1964x of the execution time required for their Python counterparts. The compiled languages of Cython and TypeScript have the most obvious performance benefits with a geometric mean time reduction of 0.3570 and 0.8363 respectively. Hack has a surprising increase of 1.0006 which suggests that adding type annotations to this test suite does little to improve its performance and may even be detrimental.

	Python	PHP	Javascript
arithmetic	0.196480938416422	1.08117249154453	0.521531100478469
strings	0.417322834645669	1	0.994459833795014
io	0.943697216140131	0.946666666666667	0.934271370062095
recursion	0.210092414068214	0.979452054794521	1.0098452883263

Table 1: Time reduction factor for untyped implementation vs typed. Values >1 imply that typed implementations are slower. Values <1 imply that typed implementations are faster.

4.1 PHP/HACK

Adams et al. [AEM+14] have determined that HHVM performance can achieve a geometric mean speedup of 2.64x within their internal benchmark suites. Results in this paper do not agree with the benchmarks of Adams et al. Figure 1a depicts the runtime (in seconds) of PHP and Hack versions. Figure 1b depicts the relative runtime where blue represents Hack and red represents PHP. Only the IO and recursive benchmarks in Hack provide a runtime advantage over PHP, and even these are within the margin of error for this experiment. Usage of Hack and HHVM do not provide any tangible benefit to performance with this test suite. It is likely that these benchmarks do not cover enough features that expose the performance of the HHVM Jit. However, the benefit of utilizing HHVM in strict mode is that it is type sound, and developers can be confident that type errors will not be encountered during runtime.



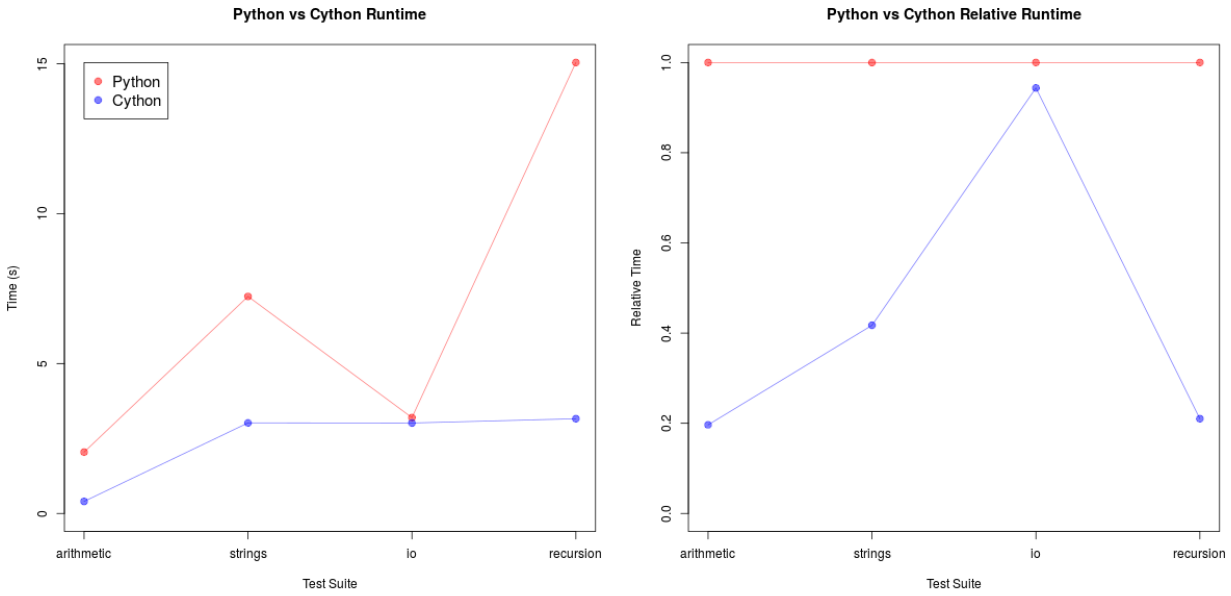
(a) PHP Typed vs Untyped Runtime

(b) PHP vs Hack Runtime Reduction

Figure 1: 1a: Depicts runtime in seconds. 1b: Depicts relative runtime. Blue refers to Hack and is scaled to 1x runtime reduction. Red values refer to PHP. Lower values are better.

4.2 Python/Cython

From the literature review on related work in Section 2, it was observed that Cython implementations can accelerate Python programs by compiling them to C code. Speedups of approximately 30x were observed in a function that implements mathematical operations such as addition, division, iterations over a loop, and multiplication. Results observed in this paper support this claim, however the speedup factor is not as extreme as observed by Wilbers et al. It is clear from Figure 2a that Cython arithmetic, string and recursive functions are all notably faster than their Python counterparts, indicated by their lower runtime (in seconds) whereas IO operations are relatively similar but still faster. Figure 2b depicts the relative runtime where blue represents Cython and red represents Python.

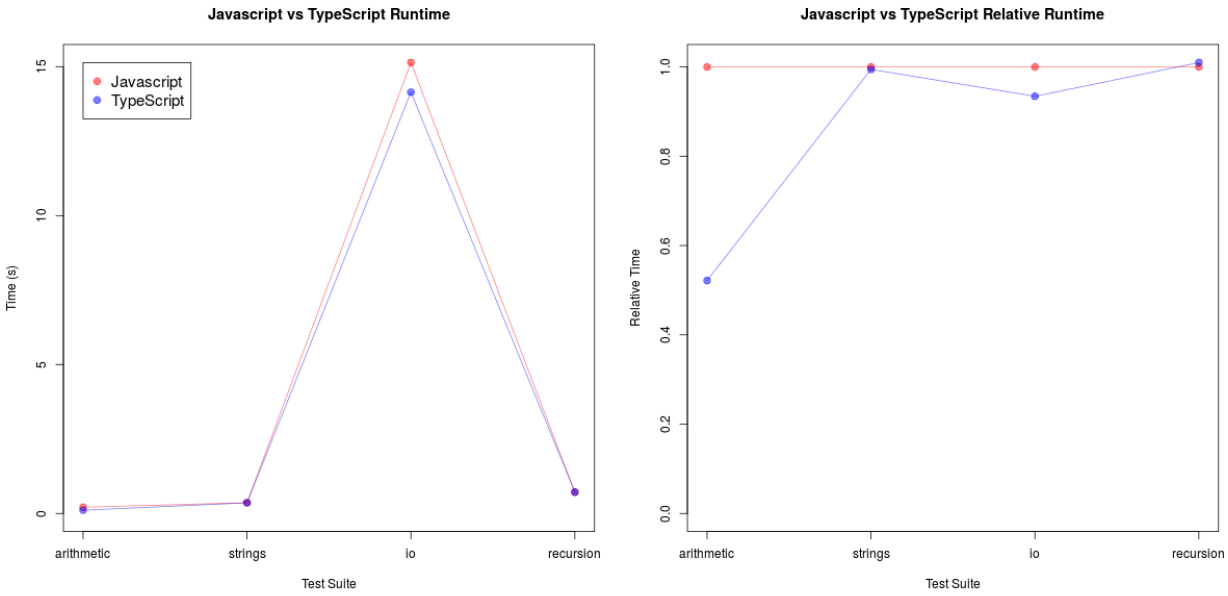


(a) Python vs Cython Runtime

(b) Python vs Cython Runtime Reduction

Figure 2: 2a: Depicts runtime in seconds. 2b: Depicts relative runtime. Blue refers to Cython and is scaled to 1x runtime reduction. Red values refer to Python. Lower values are better.

4.3 JavaScript/TypeScript



(a) JavaScript vs TypeScript Runtime (b) JavaScript vs TypeScript Runtime Reduction

Figure 3: 3a: Depicts runtime in seconds. 3b: Depicts relative runtime. Blue refers to TypeScript and is scaled to 1x runtime reduction. Red values refer to JavaScript. Lower values are better.

4.4 Threats To Validity

The primary threat to validity for this project is that the benchmarks are nowhere near comprehensive enough to encapsulate all potential use cases of each language. Results generated by this work may not be generalizable to other programming languages, or even the languages examined by this paper. This lack of generalizability is evidenced by the lack of agreement between this paper's results on HHVM and other work on JIT-compiled PHP. Another threat to validity could be that these tests are not representative of the actual runtime of functions, as functions are called an arbitrary number of times in order to generate numbers for plotting. A more sensible approach would be to incrementally increase the amount of iterations that each function is called. This would have the effect of identifying any side effects of the iteration process. Additionally, a more precise benchmarking tool could be used to measure runtimes as this would eliminate the need to iteratively call functions altogether.

4.5 Future Work

Much of the initial effort of this work went into configuring the environment within the docker container in order to properly compile and run each language. Benchmarks implemented in this paper are very rudimentary and would be improved upon by adding code that

implements inheritance relationships, complex data structures and callbacks or anonymous functions. Additionally, more complex benchmarking tools that examine characteristics such as memory usage, CPU statistics and even stack traces would be more beneficial and informative for individuals interested in researching the costs of type safety. Different configurations of the docker container could be set up in order to mimic memory swapping in a production environments.

5 Conclusion

In conclusion, the inclusion of a type system should introduce both safety benefits and performance benefits. However, boundaries between typed and untyped code may incur significant performance penalties which prevent the end-user from efficiently implementing types. This research suggests that using gradually typed programs which are statically compiled provide the most improvements to safety and performance. Future work on interpreters or (virtual machines) are promising as dynamic types may be cacheable to improve performance to levels seen in statically compiled languages [Ano17].

References

[AEM+14] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 777–790, New York, NY, USA, 2014. ACM.

[Ano17] Anon. The vm already knew that. *PACM Progr. Lang*, 2017.

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[BBC+11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[Mer14] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[RSF+15] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. *SIGPLAN Not.*, 50(1):167–180, January 2015.

[RZNV15] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for typescript. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[TFG+16] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 456–468, New York, NY, USA, 2016. ACM.

[WLØ09] Ilmar M Wilbers, Hans Petter Langtangen, and Åsmund Ødegård. Using cython to speed up numerical python programs. *Proceedings of MekIT*, 9:495–512, 2009.