

Impact of Using a Static-type System in Computer Programming

Harlin Ismail Rizky

Computer Science Department
Waseda University
Tokyo, Japan
mayharlin@fuji.waseda.jp

Hironori Washizaki

Computer Science Department
Waseda University
Tokyo, Japan
washizaki@waseda.jp

Yoshiaki Fukazawa

Computer Science Department
Waseda University
Tokyo, Japan
fukazawa@waseda.jp

Abstract—Static-type systems are a major topic in programming language research and the software industry. Static-type systems should reduce the development time and increase the code quality. Additionally, they are predicted to decrease the number of defects in a piece of code due to early error detection. However, there are only a few empirical experiments on the possible benefits of the static-type system in programming activities. This paper describes an experiment that tests whether static-type systems help developers create solutions for certain programming tasks. The results indicate that although the existence of a static-type system has no positive impact when subjects code a program from scratch, it does allow more errors in program debugging to be fixed.

Keywords—component; static-type systems, programming language, empirical study, program debugging

I. INTRODUCTION

Type systems are generally formulated as collections of rules to check the consistency of programs. This kind of checking exposes not only trivial mental slips, but also deeper conceptual errors, which are frequently manifested as type errors. A programming language can be divided into several categories as shown in Table I [1].

TABLE I
PROGRAMMING LANGUAGE CLASSIFICATION

	Statically checked	Dynamically checked
Strongly typed	ML, Haskell, Pascal (almost), Java (almost)	Lisp, Scheme
Weakly typed	C, C++	Perl

The traditional, simplified, definition of dynamic-type languages is that they do not enforce or check type safety at the compile-time (as opposed to a static-type language), but defer such checks until the run-time. While factually true, this definition leaves out what makes dynamic-type languages interesting—for example, they lower development costs and provide the flexibility required by specific domains such as data processing [3].

It should be noted that since there is no central authority defining dynamic-type languages, such languages vary greatly. Nevertheless all such languages share a great deal in common. In contrast to dynamic-type programming languages, static-type programming languages do type checking (the process of

verifying and enforcing type constraints) at the compile-time as opposed to the run-time.

There is a long, ongoing debate about the potential strengths and weaknesses of static- and dynamic-type systems in software development. Although many authors claim that static-type systems reduce the amount of time required to develop a program and consequently, improve software quality, others hold the opposite view.

Static-type checking allows early detection of some programming errors. Errors that are detected early can be fixed immediately, rather than lurking in the code to be discovered at a later time when the programmer may be busy with something else—or even after the program has been deployed [1]. When a type system is well designed, type checking can capture a large fraction of routine programming errors, eliminating lengthy debugging sessions [5].

However, current static-type systems in mainstream object-oriented languages have little expressive power. For example, although they prevent users from adding a string to a bool, they do not prevent them from accessing the first element of an empty list, creating off-by-one errors, or using null pointers. In fact, static-type systems cannot detect most common programming errors [3]. For such systems to work, developers must spoon-feed the types during their development. Additionally, any error caught by type checking will be found easily when reasoning about a specification. However, large specifications are seldom verified, and type checking can catch such errors that would otherwise go undetected [4].

This paper contributes to the discussion with a controlled experiment that empirically investigates possible conditions when developer should use static-type systems and potential advantages of using such systems. The experiment in this paper is built to test the hypothesis that static-type programming languages decrease development time and consequently enable developers to create better solutions for certain programming tasks as well as debugging certain program codes. Specifically this paper examines the following two research questions:

RQ1) How do static-type systems affect the development of specific programming tasks when developers code a program from scratch?

RQ2) How do static-type systems affect program debugging?

The experiment reveals that subjects who used a static-type system had a significant positive impact for debugging tasks, especially for encryption programs with many data types. On the other hand, when developers coded a program from scratch, a significant difference was not observed between static- and dynamic-type systems. The measurements are based on the number of requirement points that are successfully achieved and the number of fixed errors.

Section II briefly discusses related works in the area of empirical studies on type systems. Section III describes the initial considerations of the experiment and programming tasks used in the experiment as well as threats to validity. Then section IV shows the results by describing the measured data. Finally, section V concludes the paper.

II. RELATED WORKS

To the best of our knowledge, only a few works are published in the area of empirical evaluations of type systems. The first one is by Prechelt and Tichy [7], which concentrates on the impact of static-type checking in procedure arguments. Their experiment suggested that for many realistic programming tasks, type checking of interfaces improves both productivity and program quality. However, in another paper [6], which compares seven programming languages, Prechelt showed that programmers who used a scripting language (dynamic-type) needed less than half the time to finish the experimental task compared to those using a static-type language.

In a different experiment, Hanenberg [10] showed a negative impact for a static-type system in one task and no significant difference in the other. The author measured two different points: the development time required to create a minimal scanner program and the quality of the resulting software measured by the number of successful test cases. Another experiment performed by Hanenberg, which focused on the relationship between type casting and development time [11], revealed a positive impact for a dynamic-type language. However, a positive impact could not be measured for non-trivial programming tasks.

A study on the Rosetta code, which is a code repository of solutions for common programming tasks in various languages, concluded that strongly typed languages are significantly less prone to runtime failures than interpreted or weakly typed language because more errors are caught at the compile-time. Nevertheless, these works referred to run-time failures or errors that make a program terminate (including inputs that cause a program to malfunction) or unable to run rather than using a set of test cases or testing based on specific requirements (black/white box testing) [9].

A qualitative study on the Ruby programming language carried out by Daly et al. [8] suggested that, at least in the specific setting of the experiment, the benefit of the type of system could not be shown.

In a paper entitled Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects[12], the authors investigated 100,000 open source projects available

on GitHub and found no correlation between programming language employed and the number of issues listed on the bug tracker.

III. EXPERIMENT DESCRIPTION

A. Initial Considerations

Whether static-type systems reduce the development time and produce a better output remains controversial. For example, static-type systems have increased development time due to type casting. The intent of the experiment is to check whether static-type systems help programmers code a solution from scratch and debug programs as well as identify under what conditions static-type systems are beneficial with regards to the number of fulfilled requirements and the number of fixed errors measured by manually prepared test cases.

We divided the experiment into two sessions. In the first session, subjects were asked to code a program from scratch, while in the second session they were asked to fix several errors in a given program code.

The day before the experiment, the subjects were given the program requirements, which included a demo video showing how the finished program should look. Therefore, during the experiment, the subject knew what to do and what functions or procedures were necessary to complete the tasks.

B. Environment and Measurement

The subjects were allowed to select their own code editor because we assumed that using a familiar development environment would produce a better code. Although the development environment and the code editor employed in the coding activity may affect the productivity, the experimental setup minimized the impact. First, the experiment requires a relatively small number of classes and procedures. Second, the auto-complete feature is not very useful in the experimental tasks.

We measured the number of achieved requirement points and fixed errors by running several test cases on the programs created by the subjects.

C. Programming Tasks

In the first session of this experiment, 14 subjects were asked to write 2 kinds of programs: a simple validation program and an encryption program. The main difference between these two is that one involves considerably more data types and requires more type casting. Each program had 7 requirement points (features) that must be implemented. Details of the programming tasks include:

- **Simple program validation task** requires the subject to create a form with several textboxes and apply a validation to each textbox. Examples are username, password, phone number, and e-mail address validation.
- **Encryption program task** requires the subject to create a simple algorithm to encrypt and decrypt a text file and

validate whether the target file is made by the same program by placing a specific signature.

D. Experiment Execution

The experiment was performed with 14 subjects, mostly Computer Science graduates who were IT-professionals and graduate school students. Half were assigned to the static-type group and the other half to the dynamic-type group.

We divided the experiment into two sessions with a long break between the sessions. Each session was 90 minutes. The first session (coding) included two tasks: a simple validation program followed by an encryption program. Each subject had 45 minutes to complete a task and did not have a break between tasks. In the second session (debugging), the target program was similar to the program the subjects created in the first session. However, this time subjects were asked to fix errors in the given program code. Errors included a semantic error, a logical error, and a defect error related to the software requirements.

After the experiment, we asked several subjects to provide comments regarding the experiment and what was needed for a future experiment with an increased number of subjects and more diverse programming tasks.

E. Threats to Validity

As with any empirical study, this study has a number of potential threats to validity, including a small number of subjects, small programming tasks, and an artificial development environment. However, it should be emphasized that while a small programming task might not represent a real-world programming task, a large programming task has other factors that must be taken into account.

Another possible threat to validity is developer knowledge. Although we used only Computer Science graduates, we did not interview the subjects prior to experiment. Hence, there might be a gap in the subject’s coding capabilities. Nonetheless, we also realize that there is not a well-accepted standard to classify whether someone is a good or bad software developer or to indicate if one subject is equal to another.

IV. RESULT AND ANALYSIS

The experiment results and analysis are presented by giving descriptive statistics followed by significance tests to verify whether there is a significant difference between static- and dynamic-type solutions.

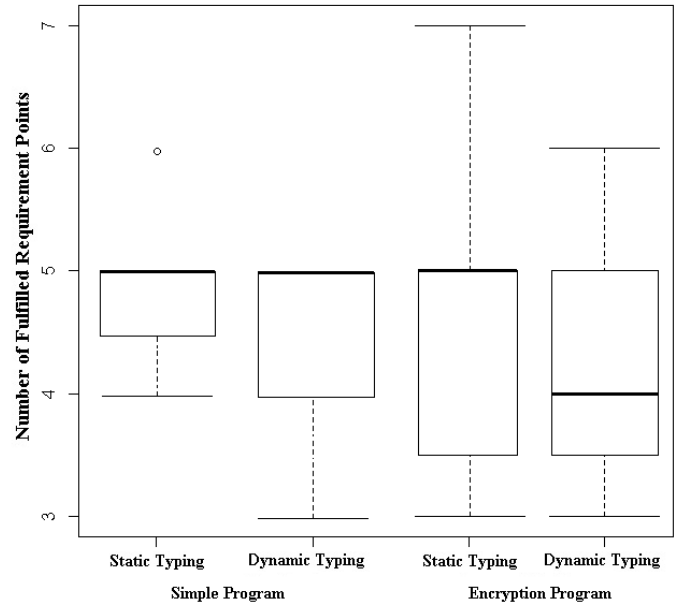


Figure 1. Number of Fulfilled Requirement Points in the Coding Session

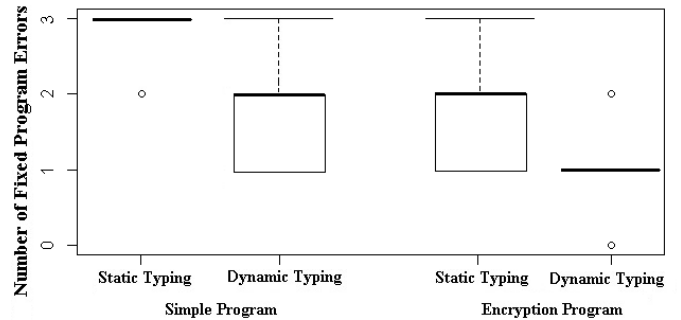


Figure 2. Number of Fixed Program Errors in the Debugging Session

A. How do static-type systems affect the development of specific programming tasks when developers code a program from scratch? (RQ1)

Figure 1 shows that a gap does not exist between solutions written in a static- and a dynamic-type system with regards to the number of achieved requirement points. This also applies to the result of the encryption program.

Furthermore, we used Wilcoxon Rank Sum Test for independent samples to determine whether there is a significant difference between the number of fulfilled requirement points by type. Since the number of samples of static- and dynamic-type solutions is equal ($n_1=n_2=7$) for both the simple validation program and the encryption program, we chose 36 [2] as the critical value (W_{crit} using $\alpha = 0.05$ two tail). Because the Wilcoxon Rank Sum values (W) are 46 and 50 for the simple validation and the encryption program, respectively (Table II), we cannot reject the null hypothesis. Thus, for code written in the scratch tasks, both type systems produce similar results.

TABLE II
WILCOXON RANK SUM VALUE OF PROGRAM SOLUTIONS

	N	Type Systems	Simple Program		Encryption Program	
			Mean Rank	Sum Rank	Mean Rank	Sum Rank
Session 1 Code from scratch	7	Static	8.43	59.0	7.86	55.0
		Dynamic	6.57	46.0	7.17	50.0
Session 2 Debugging	7	Static	9.93	69.5	10.36	72.5
		Dynamic	5.07	35.5	4.64	32.5

B. How do static-type systems affect program debugging? (RQ2)

Figure 2 shows the number of successfully fixed errors in the simple validation program. It shows that there is a visible gap between the static- and dynamic-type solutions. This gap is larger for the encryption program solutions.

Again, we used the Wilcoxon Rank Sum Test for independent samples to test whether there is a significant difference between the number of successfully fixed errors in the debugging session. Since the static- and dynamic-type solutions have the same number of samples ($n_1=n_2=7$) for both the simple validation program and the encryption program, we chose 36 [2] as the critical value (W_{crit} using $\alpha = 0.05$ two tail). The Wilcoxon Rank Sum value for the simple validation program solutions (W) is 35.5, which is slightly lower than the specified critical value. For the encryption program solutions, the W value is 32.5 (see Table II). Therefore, there is a meaningful difference in the number of fixed errors between static- and dynamic-type solutions.

In conclusion, we reveal that a static-type system enhances the effectiveness of developers in program fixing or program debugging. Nonetheless, our current data still unable to confirm whether the type of program directly affects the debugging process.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we present an experiment that explores the impact of using a static-type system in the development of certain programming tasks with respect to the number of fulfilled requirements. We also investigate whether a static-type system helps developer fix program errors. Although the tasks are considered trivial, we hope our experiment contributes to the discussion of when to use a static-type system because empirical data that can be used to identify such situations is scarce.

One interesting point is that this study weakens the argument of authors who argue that dynamic-type systems reduce the development time due to the absence of type casting. In fact, in the debugging session of our experiment, the result shows a positive impact of static-type solutions for encryption program, which involves type casting.

The result of the experiment can be summarized as follows:

- When subjects coded from scratch, there is not a significant difference in terms of the number of successfully achieved requirement points between

static- and dynamic-type solutions. This applies to both programs.

- In errors-fixing tasks, a static-type system may be beneficial. Subjects who used a static-type system tended to fix more errors. Additionally, this benefit is more pronounced in encryption programs, which contain more data types.

As a future work, we will conduct additional experiments with more subjects and more diverse programming tasks to elucidate the characteristics of software programming activities where static- and dynamic-type systems are more beneficial. This will allow software developers to select the most suitable language. We are currently investigating the possible benefits of a static-type system in large-scale software projects. However, in large-scale software development there are too many factors to take into account because there are several software development phases. As the result, the use of a static- or a dynamic-type system in programming activities may become negligible.

REFERENCES

- [1] Benjamin C. Pierce. Types and programming languages. MIT Press, Cambridge, MA, USA, 2002.
- [2] Kanji, Gopal K. 100 Statistical Tests. SAGE Publication Ltd. London. 1993.
- [3] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.
- [4] Lamport, L. and Paulson, L. C. Should your specification language be typed, vol. 21. ACM, New York, USA, 1999.
- [5] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.
- [6] Lutz Prechelt. An empirical comparison of seven programming languages, *ieee computer* (33). *Computer*, 33:23–29, 2000.
- [7] Lutz Prechelt and Walter F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Software Engineering*, 24(4):302–312, 1998.
- [8] Mark T. Daly, Vibha Sazawal, and Jeffrey S. Foster. Work in progress: an empirical study of static typing in ruby. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, Orlando, Florida, 2009.
- [9] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta Code. In *proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, pages 778–788. IEEE, 2015.
- [10] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '10*, pages 22–35, New York, NY, USA, 2010.
- [11] Stefan Hanenberg and Andreas Stuchlik. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages, DLS 2011, October 24, 2011, Portland, OR, USA*, pages 97–106. ACM, 2011.
- [12] Tegawende F. Bissyande, Ferdian Thung, David Lo, Lingxiao Jiang, Laurent Reveillere. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. *37th Annual International Computer Software & Applications Conference (COMPSAC 2013)*, pages 1–10. Kyoto, Japan, 2013.