What is the Impact of Static Type Systems on Programming Time?

Preliminary Empirical Results

Stefan Hanenberg

University of Duisburg-Essen,
Institute for Computer Science and Business Information Systems, 45117 Essen
e-mail: stefan.hanenberg@ icb.uni-due.de

Abstract

Although static type systems are an essential part in teaching and research in software engineering and computer science, there is hardly any knowledge about what the impact of type systems on the development time for a piece of software is. On the one hand there are authors that state that static types decrease an application's complexity and hence its development time. On the other hand there are authors that argue that static types increases development time since they restrict developers to express themselves in a desired way. This paper presents some preliminary results of an experiment that studies the impact of a static type system on the development speed of an application. The results reveal that the existence of the type system significantly turned out to increase the development time under the experiment's conditions.

Categories and Subject Descriptors D.1.0 [**Programming techniques**]: General

General Terms Measurement, Experimentation, Languages, Human Factors.

Keywords *Type Systems, Empirical Study, Development Time*

1. Introduction

Type systems (see for example [1, 10]) are one of the major topics in research, teaching as well as in industry. In research, new type systems appear frequently either for existing programming languages (as for example the introduction of Generics in Java) or new programming languages constructed for studying a certain type system.

In teaching, students are educated in the formal notation of type systems as well as in proofs on type systems (see for example [1, 10]). In industry, type systems become important for different reasons. Possibly a programming language in use evolves by introducing a new type system. If this new type system should be applied, developers need to be educated which produces additional costs. Maybe existing libraries or products should be adapted to match the new type system which produces additional costs as well. Finally, additional tools might be required due to the new type system (such as tools that measure the current state of the software product) which potentially produces additional costs as well.

For industry it is important to determine whether such an investment is reasonable, i.e. whether the expected benefit of type systems represents some future revenues. This means, it is necessary to understand what the advantages and maybe additional costs of using a type system are.

In industry, it is observable that untyped programming languages such as PHP or Ruby become more and more important for specific domains such as website engineering. Furthermore, untyped programming languages such as TCL or Perl are still used in software development. For future developments of such languages it seems valid to ask, whether new releases of such languages should provide a type system – assuming that a type system has a positive impact on software development.

However, while type systems are well-studied from the perspective of theoretical computer science, there is hardly any knowledge about, whether a type system plays a relevant role in the practical application of a programming language. There is even a number of researchers that advocate the use of untyped programming languages instead of typed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VASE '09 Workshop at Onward!, 26 October 2009. Orlando, Florida, USA
Copyright © 2009 ACM 1-59593-XXX-X/0X/000X…$5.00.

ones (see for example [14]) – and who emphasize the tradition of untyped programming languages such as Smalltalk.

This paper contributes to this question by a first, small evaluation of a larger data set gathered in an experiment whose focus is on type systems. Our focus in the experiment is on development time. Hence, topics such as readability, understandability and maintainability of typed / untyped programs are out of the scope of this paper.

We will show that within the experiment the use of the untyped programming language turned out to have a significant positive impact on the development speed, i.e. subjects using the untyped programming language turned out to be significantly faster than the ones using the typed programming language.

Section 2 briefly discusses frequent arguments for and against type systems. Section 3 introduces the experiment. Section 4 analyses the data. After discussing some related work in section 5, section 6 discusses and concludes this paper.

2. Typed vs. Untyped Programming Languages

In literature, there is a number of arguments for or against type systems. For example, [2] argues pro type systems as follows:

- Type systems can capture a large fraction of recurring programming errors
- Type systems have methodological advantages for code development
- Type systems reduce the complexity of programming languages
- Type systems improve the development and maintenance in security areas

On the other hand common arguments against type systems can be found for example in [4, 9, 14]:

- Type systems unnecessarily restrict the developer
- No-such-method exceptions which are caused at runtime because of missing type-checks do not occur that often
- No-such-method exceptions mainly occur because of null-pointer exceptions (which occur in typed programming languages as well)

The arguments for and against type systems seem to be valid but contradict each other. For industry, a common problem here is that it is unclear which arguments should be trusted. If for example the decision is whether the new type system of Java 1.5 should be used, additional costs for training, etc. become relevant. However, without the information whether there is any return on investment, it is hard to make a corresponding decision.

3. Experiment

The study introduced here is part of a larger experiment which focuses on the impact of type systems. While the data of the experiment is not yet completely evaluated, this paper focuses on a subset of gathered data. Section 3.1 gives an overview of the whole experiment and the part of the experiment this paper focuses on. Section 3.2 briefly describes the programming language used in the experiment. Section 3.3 describes the experiment setting, section 3.4 describes the measurement and 3.5 discusses the validity of the experiment.

3.1 Experiment Overview

In the experiment, 49 subjects (undergraduate students, selected using convenience sampling [15]) were asked to write a simplified Java parser, whereby the data used within this paper relies only on 26 subjects. The specification of the simplified Java parser to be written was delivered via a context-free grammar. The subjects had passed already fundamental Java programming courses as well as fundamental courses on formal languages.

Based on an interview the subjects were divided into two groups where the intention of the interview was to detect more experienced developers among the students and to equally divide them into the two groups with the same capabilities. Each group was trained in a programming language written for the experiment. The only difference between both languages was, that the one had a type system while the other one had not.

After training, each subject had a fix amount of time to implement the parser, whereby an additional task was to implement a simple scanner (the data collected for the scanner is in the focus of this paper).

3.2 Programming Language

For the experiment, a new object-oriented programming language (similar to Smalltalk or Ruby) called Purity was written in two versions: a typed as well as an untyped version. The reason for writing a new language was, that we wanted to exclude any influence on the experiment caused by subjects who already know the language being used. Consequently, the language was being taught as part of the experiment and the language and its documentation is not available in public (in order to prevent subjects to learn the language outside the experimental setting).

Purity is a simple object-oriented language with single inheritance and late binding. Corresponding to the language design of Smalltalk, Purity does not distinguish between primitives and objects. The language provides blocks, which is the Smalltalk version of closures. Purity contains few language constructs. Similar to the language design of Smalltalk, `if` is not a language construct itself, but is pro-

vided by corresponding methods in class Boolean. The only loop provided is the `while`-loop which is represented by a method in block objects. Field access is only provided via methods, i.e. only an object itself is allowed to access its fields. The language does not provide any constructs for multithreading.

The programming language includes a simple API with 29 basic classes such as Integer or String or LinkedList. Additional to the programming language, a small IDE was provided, which includes a class browser, a test browser (for running the application and tests of applications) and a console window.

The type system of typed Purity is non-generic and nominal and can be compared to the type system of Java up to version 1.4. The typing of blocks was implemented according to the proposal that can be found in [6].

3.3 Experimental Setting

The programming language was taught in 16 hours to the subject. After that, each subject had to implement a task. The task was to implement a Parser for a simple Java-like grammar within 27 working hours divided into 4 working days. The corresponding context-free grammar was described by the Backus-Naur notation.

These 27 hours were controlled and took place in the experimental environment. Coffee-breaks etc. were not included in these 27 hours. The subjects were permitted to arrange their time freely, i.e. they were permitted to arrive at different times. The subjects were not permitted to take any material from the experimental environment. I.e. the language itself, handbook, etc. always stayed in the experimental environment.

3.4 Measurement

As already stated, the here described paper is not based on the complete measurement of the experiment. Instead, we only rely on the first part of the experiment, the implementation of the scanner. The task was to remove characters such as white space or line feet from the input string and to create a list of tokens.

While the experiment was running, all changes in the code base were logged so that it was later on possible to reconstruct all user inputs at a certain point in time: whenever the developer added or removed a class or whenever a method was added to (or removed from) the system, a corresponding log entry was generated in the background.

For the scanner we defined a set of 15 test cases that represents the minimal functionality required in the system. These test cases were not delivered to the subjects.

Based on the log entries and the test cases we were able to determine the point in time when subjects fulfilled these test cases. We did that by reconstructing all developer entries and after any change in the code we rerun the tests. We considered the point in time when all test cases were fulfilled as the reference point where the developer finished a minimal scanner.

The data used here in the paper comes from 26 subjects, 13 from the group with the typed language and 13 from the group with the untyped language.

3.5 Threats to validity

There are some points that threaten the validity of the experiment.

Untyped		
Subject	sec	hours
1	8141	2.26
2	23041	6.40
3	2851	0.79
4	42501	11.81
5	24666	6.85
6	13975	3.88
7	7260	2.02
8	11224	3.12
9	18317	5.09
10	7335	2.04
11	16994	4.72
12	12766	3.55
13	18334	5.09

Typed		
Subject	sec	hours
14	40970	11.38
15	6305	1.75
16	32199	8.94
17	28463	7.91
18	32112	8.92
19	29509	8.20
20	28752	7.99
21	8972	2.49
22	23413	6.50
23	56985	15.83
24	3653	1.01
25	40864	11.35
26	46032	12.79

Figure 1. Experiment results

First, it can be argued that 26 hours training is not enough for learning a new language (and a new IDE). However, it should be emphasized that the language, its API as well as its' IDE was kept very simple, so that we considered the training to be sufficient. The only problematic element we identified was that the subjects had problems to understand the semantics of blocks due to the Java background of the subjects. However, since this problem was equal for all subjects, we consider it to be less problematic for the internal validity of the experiment.

	sum	max	min	arith. mean	median
untyped	57.61	11.81	0.79	4.43	3.88
typed	105.06	15.83	1.01	8.08	8.20

Figure 2. Descriptive data (in hours)

A second point is more problematic: it was not explicitly requested from the subjects that they have to finish the scanner completely before continuing with writing the parser. As a consequence, we cannot be sure at what point in time the subjects switched to the parser task without finishing the scanner task. In fact, this is the reason why we consider in this paper only 13 subjects of each group: for these subjects it was obvious that they spent the whole time on the scanner until the test cases were fulfilled.

4. Analysis

We start the analysis by describing the experiment's results and then performing significance tests in order to check whether there are significant differences in the development times using the untyped and typed programming language.

4.1 Results and Descriptive Statistics

Figure 1 shows the measured results of the experiment, i.e. the number of seconds (and hours) required by each subject to fulfill the test cases. For example, on the left hand side, the third line says that subject number 3 required 2851 seconds (respectively 0.79 hours) in order to fulfill the minimum requirements. Subject number 24 (which is a subject that used the typed language) required 3653 seconds (respectively 1.01 hours).

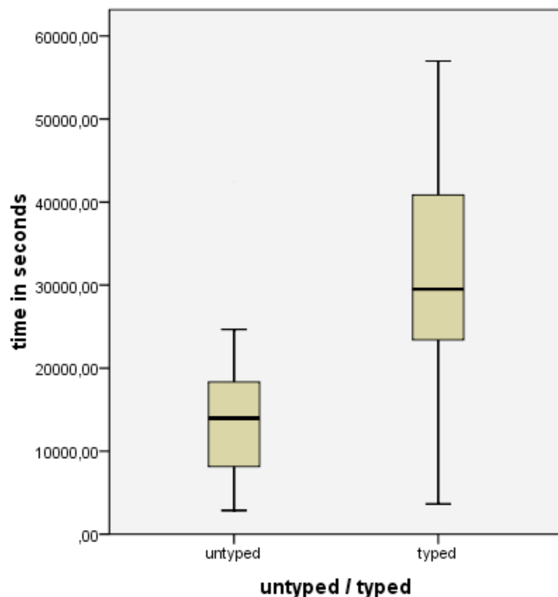

Figure 3. Box plot

Based on the experiment results, we computed the descriptive statistics (see Figure 2). Here, we see that the sum of development times (and hence the arithmetic mean), the maximum, the minimum and the median of the group with the untyped programming language is less than the corresponding values of the group with the typed programming language. Furthermore, we see that the differences are quite high (for example, the sum of development times of the untyped solution is 207,450 seconds, while the sum of times for the typed solution is 378,229 seconds).

In order to have a more obvious representation of these differences, Figure 3 shows a box plot for the untyped and typed group, which visualized the lower quartile, the median and the upper quartile of the underlying data set.

Because of these results, it seems reasonable to assume, that there is a significant difference between both times which will be checked in the next section.

4.2 Checking Significant Difference in Means

In order to check whether there is a significant difference between both values it is necessary to formulate a hypothesis which needs to be checked by a corresponding significance test.

Since the experiment design is a one factor design (with the variable typing), and since each subject solves the programming task only once either using the typed or using the untyped programming language, a significant test for independent samples is required.

The most conservative test that can be applied here is the (non-parametric) Mann-Witney U-test which does not assume the underlying data to be normally distributed (cf. [3]) and which compares for two samples whether they come from the same population.

		N	Mean Rank	Sum Ranks
Language	Untyped	13	10.23	133
	Typed	13	16.77	218

Figure 4. Ranks

The underlying hypothesis (and alternative hypothesis) to be tested is

> **H0**: The data for typed and untyped development times are drawn from the same population (i.e. equals medians)
>
> **H1**: The samples come from different populations

Figure 4 shows the resulting ranks for the sample. There, it can be seen that the mean rank for the untyped language is 10,23 while the mean rank for the typed language is 16,77. Based on this, we computed the p-value, i.e. the probability that there is no difference between the medians of development times, which is $p=0.029$. Since $p < 0.05$, the null hypothesis can be rejected, i.e. both samples come from different populations. This implies that the means of both samples significantly differ. By checking the mean ranks, it turns out that the development times using the untyped language are significant lower than the development times using the typed programming language (mean rank 10.23 in comparison to 16.77).

4.3 Computing Difference in Means

While the previous section determined that there is a significant difference in the means of the development times of typed and untyped solutions, it did not state how large this difference is (note that the use of the arithmetic mean or the median is not valid here).

	t	df	sig (2-tailed)	mean diff.	95% confidence interval	
					lower	Upper
Language	-2.524	24	0.019	-13140.3	-23887	-2393

Figure 5. T-Test results

First, we check, whether we can assume that the underlying data is normally distributed. This is done using the Shapiro-Wilk test (cf. [3]) which computes a value p. In case p is larger than 0.05, the test does not reject the assumption that the underlying data is normally distributed.

Computing the p-values reveals for the untyped programming language a value p=0.11 and for the typed programming language a value p=0.59. Hence, in both cases the test does not reject the hypothesis of a normal distribution of the underlying sample.

Hence, we can perform a t-test for independent samples which will reveal the size of the difference between both samples.

Figure 5 shows the results of the t-test. Again, we can see that there is a significant difference between both samples (with p=0,019), but we have this information already from the previous section. However, the important information is the 95% confidence interval which states that with the probability of 95% the development time using the untyped language is between 2393 and 23887 seconds less than the development time using the typed language. Hence, the advantage of using the untyped programming language in the experiment is approximately between 40 and 400 minutes.

5. Related Work

We are aware only of two works in the area of empirical evaluations of type systems. The first one by Prechelt and Tichy [13] concentrates on the impact procedure argument type checking. Prechelt and Tichy check the impact of typing by letting the subject perform a programming task in a typed as well as in an untyped language with the result that there is a significant positive impact on the developer's productivity. It is noteworthy, that the results of the experiment in [13] contradict the results of the experiment introduced here.

The second experiment we are aware of is the one performed by Gannon [5]. There, a controlled experiment was performed that compared the use of a typed vs. an untyped programming language. The experiment also revealed a positive impact of static types.

One further experiment by Prechelt [12] could be considered as an experiment that compares typed and untyped programming languages. However, the main focus of the experiment is not the typing issue (but the question whether or not the language is a script or compiler language) and there is no data available from within the development progress.

In [8] we made a first evaluation of the costs for using the untyped language based on the same data as used here in this paper. There, it turned out that between 0% and 24% of all test runs of the untyped solutions failed due to a NoSuchMethod exception. Based on these results, it is even more surprising that the untyped solutions turned out to take less time than the typed ones, because it seems clear that additional runtime costs caused by NoSuchMethodException (which do not exist in the typed solutions) slow down the development speed.

6. Conclusion and Further Work

In this paper we presented preliminary results of an experiment that explores the impact of a type system on the development time of a piece of software. We showed that within the experiment the use of the type system has a significant negative impact on the development time, i.e. the type system caused an additional overhead.

Based on an computation of the 95% confidence interval we were able to approximate the difference between both approaches and it turned out that the use of an untyped programming language took between 40 and 400 minutes less than the typed solutions. Compared to the maximum time required for the typed solution (56985 s according to Figure 1) this means that the untyped solution required between 4% and 42% less.

The interesting point in the study is that we rather assumed that developers benefit from the typed language, especially, because the language in use was new to them. Hence, we rather expected a positive impact of typing because it is commonly assumed that typing improves the ability to learn new APIs – since typing restricts a possible faulty use of a new API.

It must not be forgotten that there are some threats to validity that threaten the experiment's results. The most problematic point here is the to understand how developer differ with each other. Here, it would be more desirable to have some more objective metrics available that better permit to classify subjects.

Although this difference is quite impressive, a much deeper analysis is required on the data. First, we would like to measure the execution time of test runs for untyped solutions and compare it with the time developers required to solve typing problems – maybe the effect of the difference

is, that the developers using the type language wrote less efficient test cases. Hence, maybe the result of the experiment is mainly derived from the execution times of the applications and not from the effect of typing.

While we already made some preliminary studies about the frequency of NoSuchMethodExceptions in untyped solutions, a comparison of their execution time and the time required to solve typing errors in the typed solutions is up to future work.

Another interesting point about the experiment results is, that it contradicts studies such as [13] where a positive impact of typing could be measured. We are currently not aware of how to explain this contradiction. Possibly, the effect of typed programs is better in some situations while it is not in others. However, since the knowledge about different "kinds of programs" is currently quite restricted, it is currently unknown whether different "kinds of programs" have a different impact on the use (and the impact) of type systems.

Even though the experiment seems to suggest that typing has not a positive impact (at least not such a huge positive impact as often claimed) it must not be forgotten that the experiment has some special conditions: the experiment was a one-developer experiment. Possibly, typing has a positive impact in larger projects where interfaces between developers need to be shared.

A further point that needs to be emphasized is, that the experiment here addresses only pure programming time. Possible influence on design time, readability or maintenance of software cannot be concluded from the experiment's results.

While the discussion about typing mainly concerns the design of languages in known areas such as object-oriented programming or functional programming, our intention is in the future to study the impact of typing in newer approaches such as aspect-oriented programming. We did a first evaluation of aspect-oriented programming already in [7], but did not consider any typing effects there.

A general conclusion is that definitively more studies are needed in order to determine the impact of type systems. Since type systems play such an important role in software development it is rather tragic that the discussion about the need for typed and untyped programming languages is mainly driven by personal experiences and personal preferences of researchers instead of empirical knowledge received from an adequate research method.

References

[1] Bruce, K.: Foundations of Object-Oriented Languages: Types and Semantics, MIT PRESS, 2002

[2] Cardelli, Luca: Type Systems, In: CRC Handbook of Computer Science and Engineering, 2nd Edition, CRC Press, 1997.

[3] Conover, W. J.: Practical nonparametric statistics, 3rd edition, John Wiley & Sons, 1998.

[4] Eckel, B.: Strong Typing vs. Strong Testing, mindview, 2003, http://www.mindview.net/WebLog/log-0025, last access: August 2009

[5] Gannon, J. D.: An Experimental Evaluation of Data Type Conventions, Comm. ACM 20(8), 1977, S. 584-595.

[6] Graver, J. O.; Johnson, R. E.: A Type System for Smalltalk, Seventeenth Symposium on Principles of Programming Languages, 1990, pp. 136-150

[7] Hanenberg, S., Kleinschmager, S., Josupeit-Walter, M: Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. Accepted for publication at 3rd International Symposium on Empirical Software Engineering and Measurement, 2009.

[8] Hanenberg, S.: Costs of Using Untyped Programming Languages – First Empirical Results. 13th IFAC Symposium on Information Control Problems in Manufacturing (Track Advanced Software Engineering). Moskow, June 3-5 2009.

[9] Lamport, L.; Paulson, L. C.: Should your specification language be typed, vol. 21, ACM, New York, NY, USA. 1999.

[10] Pierce, B.: Types and Programming Languages, MIT Press, 2002.

[11] Prechelt, L.: Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik, Springer-Verlag, 2001.

[12] Prechelt, Lutz: An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, March 2000.

[13] Prechelt, L., Tichy W.: A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking, IEEE Transactions on Software Engineering 24(4), 1998, S. 302-312.

[14] Tratt, L., Wuyts, R.: Dynamically Typed Languages. IEEE Software 24(5), 2007, S. 28-30.

[15] Wohlin, C., Runeson, P., Höst, M.: Experimentation in Software Engineering: An Introduction, Springer, 1999