An Experiment to Assess the Benefits of Inter-Module Type Checking

Lutz Prechelt (prechelt@ira.uka.de)
Walter F. Tichy (tichy@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany

Abstract

This paper reports on an experiment to assess the error detection capabilities of static, inter-module type checking. Type checking is considered an important mechanism for detecting programming errors, especially interface errors.

The experiment uses Kernighan&Ritchie C and ANSI C. The relevant difference is that the ANSI C compiler checks module interfaces (i.e., the parameter lists of calls to external functions), whereas K&R C does not. The experiment employs a counterbalanced design, in which each subject writes two non-trivial programs that interface with a complex library (Motif). Each subject writes one program in K&R C and one in ANSI C. The input to each compiler run is saved and manually analyzed for errors.

Results indicate that delivered ANSI C programs contain significantly fewer interface errors than delivered K&R C programs. Furthermore, after subjects have gained some familiarity with the interface they are using, ANSI C programmers remove errors faster and are more productive (measured in both time to completion and functionality implemented).

1. Introduction

Datatypes are an important concept in programming languages. A datatype is an interpretation applied to a datum, which otherwise would just be a string of bits. Datatypes are used to model the data space of a problem domain and are an important aid to programming and program understanding. A further benefit is type checking: A compiler or interpreter can determine whether a data item of a certain type is permissible in a given context, such as an expression or statement. If it is not, then the compiler has detected an error in the program. It is the error detection capability of type checking that is of interest in this paper.

There is some debate over whether dynamic type checking is preferable to static type checking, how strict the type checking should be, and whether explicitly declared types are more helpful than implicit ones. However, it seems that the benefits of type checking are virtually undisputed. Modern programming languages have evolved elaborate type systems and checking rules. In some languages, such as C, the type checking rules were even strengthened in later versions. Furthermore, type theory is an active area of research.

However, it seems that the benefits of type checking are largely taken on faith or are based on personal anecdotes. For instance, Wirth states in [9] that the type checking facilities of Oberon had been most helpful in evolving the Oberon system. Many programmers can recall instances when type checking did or could have helped them. However, we could not find any reports in the literature on controlled, repeatable experiments that test whether type checking has any positive (or negative) effects. The cost and benefits of type checking are far from clear, because type checking is not for free: It requires effort on behalf of the programmer in providing type information. Furthermore, there is some evidence that inspections might be more effective in finding errors than compilers [5].

We conclude that the actual costs and benefits of type checking are largely unknown. This situation seems to be at odds with the importance assigned to the concept: Languages with type checking are widely used and the vast majority of practicing programmers are affected by the technique in their day-to-day work. The purpose of this paper is to provide initial, "hard" evidence about the effects of type checking. We

describe a repeatable and controlled experiment that confirms some positive effects: First, when applied to interfaces, type checking reduces the number of errors remaining in delivered programs. Second, when programmer use a familiar interface, type checking helps them remove errors more quickly and increases their productivity.

Definitive knowledge about positive effects of type checking can be useful in two ways: First, we still lack a useful scientific model of the programming process. Such a model is a prerequisite for understanding the overall software production process. Understanding the types, frequencies, and circumstances of programmer errors is an important ingredient of such a model. Second, there are still many environments, where type checking is missing or incomplete, and such knowledge will produce pressure to close these gaps. For instance it may pay off to invest in discriminating between the many kinds of integer values that occur in interfaces, such as cardinal numbers, indices, differences, etc.

To determine whether and to which extent such discrimination might be useful is a typical example of a software process improvement question that can be answered by defining and applying the appropriate metrics. Since full in-process measurement for this question might be quite expensive, a go/no-go decision should be made first and can be attempted by a small-scale, controlled experiment. Our study is an example of using the experimental method as the first step of a process improvement.

2. Related work

Work on error classification and detection obviously has a bearing on our experiment. Publications [2, 8] describe and analyze the typical errors in programs written by novices. The results are not necessarily relevant for professional programmers. Furthermore, type errors do not play an important role in these studies.

Error type analyses have also been performed in larger scale software development settings. Type checking has not been a concern in these studies, but in some cases related information can be derived. For instance, reference [1] reports that 39 percent of all errors in a 90.000 line FORTRAN project were interface errors. We conjecture that some proportion of these could have been found by type checking.

The error detection capabilities of testing methods is a question that has attracted considerable interest, see for instance[3]. The errors found by testing are those that already passed the type checks, so the results from these studies are hardly applicable here.

Several studies have compared the productivity effects of different programming languages. Most of these studies used programmers with little experience and very small programming tasks, e.g. [2]. Others used somewhat larger tasks and experienced programmers, but lacked proper experimental control, as in [4]. All of these studies have the inherent problem that they are confounded by too many factors to draw conclusions regarding type checking.

We are aware of only one closely related experiment, the Snickering Type Checking Experiment[6] with the Mesa language. In that work, compiler-generated error messages involving types were diverted to a secret file. A programmer working with this compiler on two different programs was shown the error messages after he had finished the programs and was asked to estimate how much time he would have saved had he seen the messages right away. Interestingly, the programmer had independently removed all the errors detected by the type checker. He claimed that on one program, which was 100% his own work, type checking would not have helped appreciably. On another program which involved interfacing to a complicated library, he estimated that type checking would have saved 50% of total development time. It is obvious that this type of study has many flaws. But to our knowledge it was never repeated in a more controlled setting.

It appears that the cost and benefits of type checking have not been studied systematically.

3. Design of the Experiment

The idea behind the experiment is the following: Let experienced programmers solve programming problems involving a complex library. To control for the type-checking/no-type-checking variable, let every subject solve one problem with K&R C, and another with ANSI C. Save the inputs to all compiler runs for later error analysis.

A number of observations regarding the realism of the setup are in order. A simple task means that the difficulties observed will stem from using the library, not from solving the task itself. Thus, most errors will occur when interfacing to the library, where the effects of type checking are thought to be most pronounced. Furthermore, using a complex library is similar to the development of a module within a larger project, where many imported interfaces must be handled. We used experienced programmers familiar with the programming language, so the results would not be confounded by problems with the language. However, the programmers had no experience with the library — another similarity with realistic software development,

where new modules are written within a relatively foreign context.

In order to balance for both learning effects and inter-subject ability differences, we used a counterbalanced design: There were two independent problems to be solved (A and B, as described below) and two treatments (ANSI C and Kernighan/Ritchie C). Each subject had to solve both problems, each with a different language. Thus, there are two experimental groups: Group 1 solves A(ANSI)+B(KR) (in this order) and group 2 solves B(ANSI)+A(KR).

Controlling for the sequence of problems and languages creates another two groups, see Table 1. The dependent variables are described in section 3.4

Subjects were assigned to groups in round-robin fashion.

3.1. Tasks

Problem A (2×2 Matrixinversion): Open a window with four text fields arranged in a 2 by 2 pattern plus an "Invert" and a "Quit" button. See figure 1. "Quit" exits the program and closes the window. The text fields represent a matrix of real values. The values can be entered and edited. When the "Invert" button is pressed, replace the values by the coefficients of the corresponding inverted matrix, or print an error message if the matrix is not invertible. The formula for 2×2 matrix inversion was given.

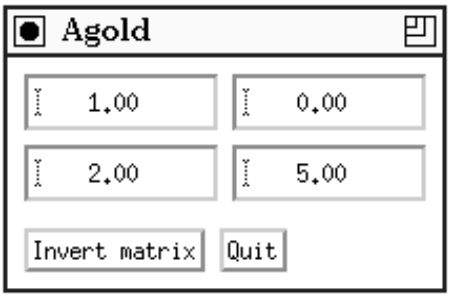

Figure 1. Problem A (2×2 matrix inversion)

Problem B (File Browser): Open a window with a menubar containing a single menu. The menu entry "Select file" opens a file selector box. The entry "Open selected file" pops up a separate, scrollable window and displays the contents of the file previously selected in the file selector box. "Quit" exits the program and closes all its windows. See figure 2.

For solving the tasks, the subjects did not use native Motif, but a special wrapper library. This library has operations similar to those of Motif, but with improved type checking (e.g. there were no variable argument lists, and resources were typed). There was also some simplification through additional convenience functions. For instance, there was a function for creating a RowColumnManager and setting its orientation and packing mode in one call.

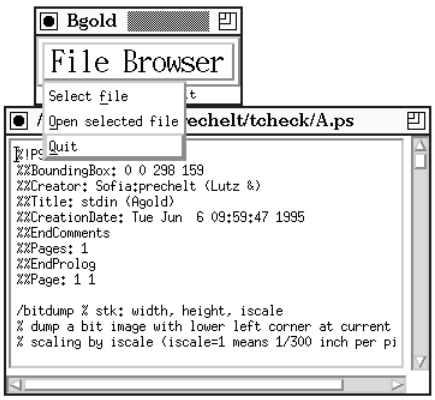

Figure 2. Problem B (File browser)

The tasks, although quite small, were not at all trivial. The subjects had to understand several important concepts of Motif programming (such as widget, resource, and callback function). They had to learn to use them from abstract documentation only, without example programs (as is typically the case in practice).

3.2. Subjects

40 unpaid volunteers participated in the study. Six of them were removed from the sample: One deleted his protocol files, one was obviously inexperienced (took almost ten times as long as the others), and four worked only on one of the two problems. After this mortality, the A/B groups had 8+8 subjects, the B/A groups had 11+7 subjects. We consider this to be still sufficiently balanced.

The remaining 34 subjects had the following education. Two were postdocs in computer science; 19 were PhD students in computer science and had completed a M.S. degree in CS; another subject was also a CS PhD student but held a M.S. in physics; twelve subjects were CS graduate students with a B.S. in computer science.

The 34 subjects had between four and 19 years of programming experience ($\mu = 10.0$) and all but eleven of them had written at least 3000 lines in C (all but one at least 300 lines). Only eight of the subjects had any programming experience with X-Windows or Motif; only three of them had written more than 300 lines in X-Windows or Motif.

	first problem A second problem B	first problem B second problem A
first ANSI C then K&R C	Group 1	Group 2
first K&R C then ANSI C	Group 3	Group 4

Table 1. Tasks and compilers assigned to the four groups of subjects

3.3. Setup

Each subject received two written documents and one instruction sheet and was then left alone at a Sun-4 workstation to solve the two problems. The subjects were told to use roughly one hour per problem, but no time limit was enforced. Subjects could stop working even if the programs were not operational.

The instruction sheet was a one-page description of the global steps involved in the experiment: "Read sections 1 to 3 of the instruction document; fill in the questionnaire in section 2; initialize your working environment by typing `make TC1`; solve problem A by..." and so on. The subjects obtained the following materials, most of them both on paper and in files:

1. a half-page introduction to the purpose of the experiment;
2. a questionnaire about the background of the subject;
3. specifications of the two tasks plus the program skeleton for them;
4. an introduction to Motif programming (1 page) and some useful commands (for example how to to search manuals on line);
5. a manual listing first the names of all types, constants, and functions that might be required, followed by descriptions of each of them including the signature, semantic description, and several kinds of cross references. The document also included introductions to the basic concepts of Motif and X-Windows. This manual was hand-tailored to contain all information required to solve the tasks and hardly anything else;
6. a questionnaire about the experiment (to be filled in at the end).

Subjects could also execute a "gold" program for each task. A gold program solved its task completely and correctly and could be used as a backup for the verbal specifications. Subjects were told to write programs that duplicated the behavior of the gold programs.

The subjects did not have to write the programs from scratch. Instead, they were given a program skeleton that contained all necessary `#include` commands, variable and function declarations, and some initialization statements. In addition, the skeleton contained pseudocode describing step by step what statements had to be inserted to complete the program. The subjects' task was to find out which functions they had to use and which arguments to supply. Almost all statements were function calls.

The following is an example of a pseudostatment in the skeleton.

```
/* Register callback-function 'button_pushed'
for the 'invert' button with the number 1 as
'client_data' */
```

It can be implemented thus:

```
XtAddCallbackF(invert, XmCactivateCallback,
button_pushed, (XtPointer)1);
```

There were only few variations possible in the implementation of the pseudocode.

The programming environment captured all program versions submitted for compilation along with a time stamp and the messages produced by the compiler and linker. A timestamp for the start and the end of the work phase for each problem was also written to the protocol file.

The environment was set up to call the standard C compiler of SunOS 4.1.3 using the command `cc -c -g` for the Kernighan/Ritchie tasks and version 2.7.0 of the GNU C compiler using `gcc -c -g -ansi -pedantic -W -Wimplicit -Wreturn-type` for the ANSI C tasks.

3.4. Observed variables

After the experiment was finished, each program version in the protocol files was annotated by hand. Each different programming error that occurred in the programs was identified and given a unique number. For instance, for the call to `XtAddCallbackF` shown above, there were 15 different error numbers, including 4 for wrong argument types, 4 for wrong argument objects with correct type, and another 7 for more specialized errors.

Each program version was annotated with the errors introduced, removed, or changed (without correcting

them).

Additional annotations counted the number of type errors, other semantic errors, and syntactic errors that actually provoked one or more error messages from the compiler or linker. The timestamps were corrected for pauses that lasted more than 10 minutes. Summary statistics were computed, for which each error was classified into one of the following categories:

comp: Errors that had to be removed before the program would pass the compiler and linker, even for K&R C. This class will be ignored.

slight: Errors resulting in slightly wrong functionality of the program, but so minor that the programmers probably felt no need to correct them. This class will be ignored.

invis: Errors that are *invisible*, i.e., they do not compromise functionality, but only because of unspecified properties of the library implementation. Changes in the library implementation may result in a misbehaving program. Example: Supplying the integer constant `PACK_COLUMN` instead of the expected boolean value `True` works correctly, because (and as long as) the constant happens to have a non-zero value. This class of errors will be ignored.

invisD: same as *invis*, except that the errors will be detected by ANSI C parameter type checking (but not by K&R C).

severe: Errors resulting in significant deviations from the prescribed functionality.

severeD: same as *severe*, except that the errors will be detected by ANSI C parameter type checking (but not by K&R C).

These categories are mutually exclusive. Unless otherwise noted, the error statistics discussed below are computed based on the sum of *severe* , *severeD* , and *invisD* .

Other metrics observed were the number of compilation cycles (versions) and time to completion, i.e., the time taken by the subjects before delivering the program (whether complete and correct or not).

From these metrics and annotations, additional statistics were computed. For instance the frequency of error insertion and removal, the number of attempts made before an error was finally removed, the time an error remained in the program ("lifetime"), and the number and type of errors remaining in the final program version.

For measuring productivity and unimplemented functionality, we define a *functionality unit (FU)* to be a single statement in the gold programs. Using the gold programs as a reference normalizes the cases where subjects used more than one statement instead. FUs are thus a better measure of program volume than lines of code. Gold program A has 16 FUs, B has 11.

We also annotated the programs with the number of *gaps*, i.e., the number of missing FUs. A FU is counted as missing if a subject made no attempt to implement it. From this, it is easy to derive the number of FUs implemented in a program.

3.5. Internal and external validity

The following problems might threaten the internal validity of the experiment, i.e., the correctness of the observations:

1. Error messages produced by the two compilers might differ for the same error, and this might influence productivity. Our subjective judgement here is that the error messages of both compilers are comparable in quality, at least for the purposes of this experiment.
2. There may be annotation errors. To insure consistency, all annotations were made by the same person. The annotations were cross-checked first with a simple consistency checker and then some of them manually. The number of annotation errors found in the manual check was negligible (4%).

The following problems might limit external validity of the experiment, i.e., the generalizability of our results:

1. The subjects were not professional software engineers. However, they were quite experienced programmers and held degrees (many of them advanced) in computer science.
2. The results may be domain-dependent. This objection cannot be ruled out. This experiment should therefore be repeated in domains other than graphical user interfaces.
3. The results may not apply to situations were the subjects are very familiar with the interfaces used.

Despite these problems, we expect that the scenario chosen in the experiment is nevertheless similar to many real situations with respect to type checking errors.

4. Results and Discussion

Most of the statistics of interest in this study have clearly non-normal distributions and sometimes severe outliers. Therefore, we present medians (to be precise: an interpolated 50% quantile) rather than arithmetic means. Where most of the median values are zero, higher quantiles are given.

The results are shown in Table 2. There are 13 different statistics in three main columns. The first column shows the statistics for both tasks, independent

	Statistic	both tasks ANSI	both tasks K&R	1st task ANSI	1st task K&R	2nd task ANSI	2nd task K&R
1	hours to completion	1.3	1.35	1.6	1.6	0.9	1.3
	$p =$	0.49		0.83		**0.018**	
2	#versions	15	16	19	21	12.5	13
	$p =$	0.84		0.63		0.16	
3	#type error messages/hour	6.3	1.1	4.3	1.2	7.7	1.0
	$p =$	**0.0000**		**0.0007**		**0.0006**	
4	#error insertions/hour	5.6	6.5	4.0	4.2	6.3	6.8
	$p =$	0.35		0.28		0.75	
5	#error removals/hour	4.15	3.95	4.0	4.2	4.9	3.7
	$p =$	0.69		0.97		0.60	
6	sum of accumulated error lifetime	1.6	2.55	2.2	3.6	0.8	2.2
	$p =$	**0.035**		0.26		**0.025**	
7	#right, then wrong again (75% quant.)	1.0	1.0	1.0	1.0	0.0	1.0
	$p =$	0.12		0.82		**0.009**	
8	#remaining errs in delivered program	1.0	2.0	1.0	2.0	1.0	2.0
	$p =$	**0.016**		0.32		**0.031**	
9	— for *invisD* only (90% quantile)	0.0	1.0	0.0	1.4	0.0	0.0
	$p =$	**0.04**		**0.048**		0.41	
10	— for *severe* only	1.0	1.0	1.0	0.0	1.0	1.0
	$p =$	0.66		0.74		0.65	
11	— for *severeD* only	0.0	1.0	0.0	1.0	0.0	1.0
	$p =$	**0.0001**		**0.015**		**0.0022**	
12	#gaps (75% quantile)	0.25	0.0	1.5	0.0	0.0	0.0
	$p =$	0.35		0.26		0.70	
13	FU/h	8.6	9.7	7.21	8.5	12.8	10.7
	$p =$	0.93		0.31		0.061	

Table 2. Medians (or other quantiles as indicated) of statistics for ANSI C vs. K&R C versions of programs and p-values for statistical significance of Wilcoxon Rank Sum Tests of the two. Values under 0.05 indicate significant differences of the medians. Column pairs are for 1st+2nd, 1st, and 2nd problem tackled chronologically by each subject, respectively. All entries include data points for both problem A and problem B.

of order. The second and third columns reflect the observations for those tasks that were tackled first and second, respectively. These columns can be used to assess the learning effect. Each main column reports the medians (or higher quantiles where indicated) for the tasks programmed with ANSI C and K&R C plus the p-value. The p-value is the result of the Wilcoxon Rank Sum Test[1] and can be interpreted as the probability that the observed differences occurred by chance. A difference in the median is considered significant if $p \leq 0.05$. Significant results are marked in boldface in the table. When the p-value is not significant, nothing can be said, i.e., there may or may not be a difference.

The first statistic, time to completion, shows that there is no significant difference between ANSI C and K&R C for the first task and both tasks together. The combined time spent for the second task is shorter than for the first ($p = 0.0012$, not shown in the table), indicating a learning effect. In the second task, ANSI C is significantly more productive. A plausible explanation is as follows. When they started, programmers did not have a good understanding of the library and where struggling more with the concepts than the interface itself. A lack of understanding is also born out by the protocols. Type checking is unlikely to help gain a better understanding. Type checks became useful only after programmers had overcome the initial learning hurdle.

Statistic 2, the number of program versions compiled, does not show a significant difference. Entry 3 shows that the ANSI C compiler does indeed flag type errors significantly more often than the K&R compiler does. Each type error was counted only once per compilation for this statistic, no matter whether it produced one or several messages. Messages produced for other semantic or for syntactic errors were ignored.

Entries 4 to 7 are statistics that describe the internal error processes, all based on the sum of the error categories *invisD*, *severe*, and *severeD*. The frequency of error insertion and removal (entries 4 and 5) show no significant differences. The other two show some advantage for ANSI C and it is again most pronounced in task 2, confirming the learning effect.

The total lifetime of all errors during programming (entry 6) is shorter for ANSI C overall and in the 2nd

[1] This test, also known as Mann-Whitney U Test, was chosen because the distributions of the variables are more or less logistic, rather than, say, normal or double exponential.

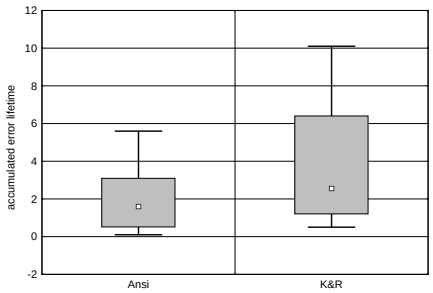

Figure 3. Boxplots of accumulated error lifetime (in hours) over both tasks for ANSI C (left boxplot) and K&R C (right boxplot). The upper and lower whiskers mark the 95% and 5% quantiles, the upper and lower edges of the box mark the 75% and 25% quantiles, and the dot marks the 50% quantile (median). All other boxplots following below have the same structure.

task. The distributions of accumulated lifetime over both tasks are also shown as boxplots in Figure 3. As we see, the K&R total error lifetimes are usually higher and spread over a much wider range.

The number of errors introduced in previously correct or repaired parts of a program (entry 7) is significantly higher for K&R C in the 2nd task.

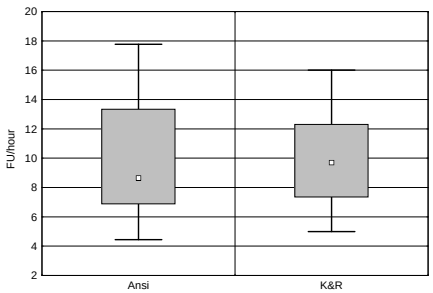

Figure 4. Boxplots of productivity (in FU/hour) over both tasks.

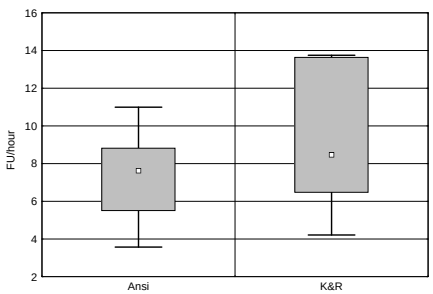

Figure 5. Boxplots of productivity (in FU/hour) for first task.

There are no significant differences in the number of gaps in the delivered programs (entry 12). However,

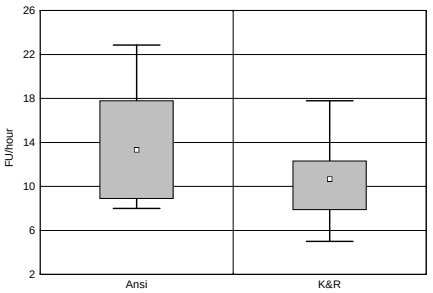

Figure 6. Boxplots of productivity (in FU/hour) for second task.

the p-value of 0.061 for productivity (entry 13) in the second task strongly suggests that ANSI C is helpful for programmers after the initial interface learning phase. The combined (both languages) productivity rises significantly from the 1st task to the 2nd task ($p = 0.0001$, not shown in the table); this was also reported by the subjects.

The distributions of productivity measured in FU/hour are shown in Figures 4 to 6. We see that ANSI C makes for a more pronounced increase in productivity from the first task to the second than does K&R C.

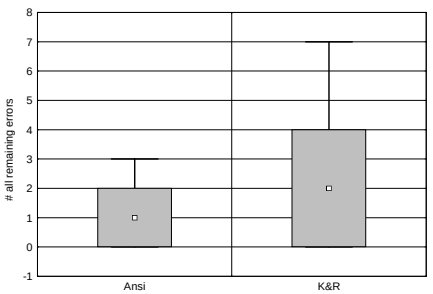

Figure 7. Boxplots of total number of remaining errors in delivered programs over both tasks.

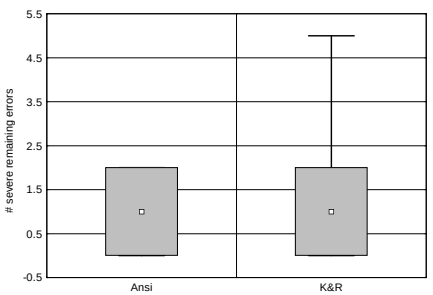

Figure 8. Boxplots of number of remaining *severe* errors in delivered programs over both tasks.

A clear advantage of ANSI C over K&R C is the number of errors still present in the delivered program (entry 8). As entries 9 to 11 indicate, this result stems

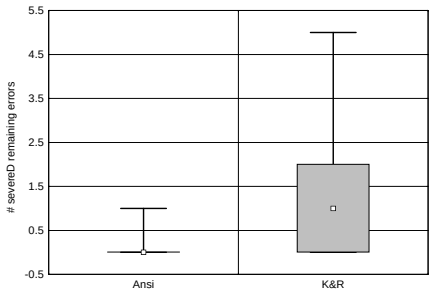

Figure 9. Boxplots of number of remaining *severeD* errors in delivered programs over both tasks.

from the direct detection of errors through type checking; little or no reduction of non-detectable errors (entry 10) is achieved. The both-task distributions for entries 8, 10, and 11 are shown in Figures 7 to 9. As we see there, the distributions for *severe* errors differ only in the upper tail (Figure 8), whereas the distributions for the *severeD* errors differ dramatically in favor of ANSI C (Figure 9), resulting in a significant overall advantage for ANSI C (Figure 7).

A detailed analysis of the errors remaining in the delivered programs indicates a slight, but not statistically significant tendency that other frequent errors also were reduced in the ANSI programs: using the wrong variable as a parameter or an assignment target ($p = 0.28$) or using a wrong constant value as a parameter ($p = 0.35$).

There were no significant differences between the two tasks. All of the above results hardly change if one considers the tasks A and B separately (not shown).

Finally, the subjective impressions of the subjects as reported in the questionnaires are as follows. 26 of the subjects (79%) noted a learning effect from the first program to the second. 9 subjects (27%) reported that they found the ANSI C type checking very helpful, 11 (33%) found it considerably helpful, 4 (12%) found it almost not helpful, 5 (15%) found it not at all helpful. 4 subjects could not decide, 1 questionnaire was lost.

5. Conclusions and further work

Our experiment confirms the following hypotheses:

1. Type checking can reduce the number of interface errors in delivered programs.
2. When using an interface, type checking can increase productivity, provided the programmer has gained a basic understanding of the interface.

One must be careful generalizing the results of this study to other situations. For instance, the experiment is unsuitable for determining the proportion of interface errors in an overall mix of errors, because is was designed to prevent errors other than interface errors. Hence it is unclear how large the differences will be if error classes such as declaration errors, initialization errors, algorithmic errors, or control flow errors are included.

Nevertheless, the experiment suggests that for many realistic programming tasks, type checking of interfaces improves both productivity and program quality. Furthermore, some of the resources otherwise expended on inspecting interfaces might be allocated to other tasks.

Further work should repeat similar error analyses in different settings (e.g. tasks with complex data flow or object-oriented languages). In particular, it would be interesting to compare productivity and error rates under compile-time type checking, run-time type checking, and type inference. Other important questions concern the influence of a disciplined programming process such as the Personal Software Process [5]. Finally, an analysis of the errors occurring in practice might help devise more effective error detection mechanisms.

Acknowledgements

Thanks to our experimental subjects. Many thanks in particular to Paul Lukowicz for patiently guinea-pigging the experimental setup.

References

[1] V. R. Basili and B. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, Jan. 1984.

[2] A. Ebrahimi. Novice programmer errors: Language constructs and plan composition. *Intl. J. of Human-Computer Studies*, 41:457–480, 1994.

[3] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. on Software Engineering*, 19(8):774–787, August 1993.

[4] P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs. awk vs. ...an experiment in software prototyping productivity. Technical report, Yale University, Dept. of CS, New Haven, CT, July 1994.

[5] W. Humphrey. *A Discipline of Software Engineering*. Addison-Wesley, 1995.

[6] J. H. Morris. The snickering type checking experiment. unpublished, 1978.

[7] E. Soloway and S. Iyengar, editors. *Empirical Studies of Programmers*. Ablex Publishing Corp., Norwood, NJ, June 1986. (The papers of the First Workshop on Empirical Studies of Programmers, Washington D.C.).

[8] J. G. Spohrer and E. Soloway. Analyzing the high frequency bugs in novice programs. In *[7]*, pages 230–251, 1986.

[9] N. Wirth. Gedanken zur Software-Explosion. *Informatik Spektrum*, 17(1):5–20, February 1994.