Language Design for Reliable Software
S.L. Graham Editor

An Experimental Evaluation of Data Type Conventions

J.D. Gannon
University of Maryland

The language in which programs are written can have a substantial effect on the reliability of the resulting programs. This paper discusses an experiment that compares the programming reliability of subjects using a statically typed language and a "typeless" language. Analysis of the number of errors and the number of runs containing errors shows that, at least in one environment, the use of a statically typed language can increase programming reliability. Detailed analysis of the errors made by the subjects in programming solutions to reasonably small problems shows that the subjects had difficulty manipulating the representation of data.

Key Words and Phrases: data types, experimentation, language design, redundancy, reliable software
CR Categories: 4.22

A version of this paper was presented at the SIGPLAN/SIGOPS/SICSOFT Conference on Language Design for Reliable Software, Raleigh, N.C., March 28-30, 1977.

Authors's address: Department of Computer Science, University of Maryland, College Park, Maryland 20742.

This work was supported by grants from the National Science Foundation (MCS76-23346) and the National Bureau of Standards (5-9017).

1. Data Types and Programming Reliability

The goal of reliable programming is to minimize the number of errors in completed programs. Attaining this goal may involve reducing the number of errors committed by programmers and/or increasing the fraction of errors that are detected and corrected before the production of a final program. Appropriate language design can contribute to both of these goals.

One language feature which is common to many programming languages but appears in different forms in different languages is the method of determining the type of an operand. Data types may be associated with operands in one of three ways: statically, dynamically, or not at all. In a statically typed language (e.g. Pascal [8]), a data type is associated with an identifier in a declaration. During its lifetime, the identifier may only be assigned values of the same type. In a language like GEDANKEN [5] or SNOBOL, data types are associated with an operand dynamically during execution. The type of an operand is the type of the last value assigned to it. BCPL [6] and BLISS [9] are called "typeless" languages since each operand is considered to be a collection of bits (usually a word in memory). When an operator is applied to operands, the operands are assumed to represent a value of the type that the operator may manipulate. Many languages do not fit neatly into one of the three categories. For example, Algol 60 and PL/I are primarily statically typed languages. However, Algol 60 does not require complete specification of the arguments of procedures, and the UNSPEC operation of PL/I allows the type checking mechanism to be subverted.

Some language designers strongly advocate separate specification of the types of operands, while other designers feel that this practice interferes with the programming task. Those designers who advocate separate specification of data types believe that data types offer two principal advantages to a programmer: power and redundancy. The power of data types allows a programmer to think in terms of his application rather than of the characteristics of the machine on which his application will run and to use the operations defined for each type rather than building these operations out of more primitive operations. Thus the programmer deals with characters rather than bit patterns within a word and with matrices rather than storage structures of words. The programmer is aided in reaching a solution and prevented from writing code that depends upon a particular representation of his data and thereby pre-

cludes redesign of the program. Languages such as CLU [4] and Alphard [10] are being designed to offer programmers the benefits accruing from limiting the availability of information about the representation of data. Type checking prevents a programmer from attempting operations on operands which are not valid representations of the operands he wishes to process (e.g. addition of Boolean values). In statically typed languages, the context of each appearance of an operand implies a type which can be checked against its declared type. Furthermore, this checking can be performed at compile time. This redundancy is not present in dynamically typed or "typeless" languages. Errors caused by operators having operands of the wrong type can only be detected at run time and then only if the code containing the error is executed, a condition that cannot be assured by testing.

However, even among the advocates of separate specification of type, there is little agreement on the primary benefit of including this feature in a language. Some designers feel that the power of data types helps programmers avoid errors. Others believe that as many as 50 percent of their programming errors are detected by the type-checking mechanism.

As with many other issues in programming language design, there is a wealth of personal experience with data types, and every language designer feels that his experience is the valid one. We cannot logically prove that particular language features will enhance programming reliability, much less derive the amount of improvement by analysis. However, we can gather empirical evidence that tends to confirm or refute such claims by measuring the amount of improvement (or lack thereof) in real situations. We can observe programmers at work and examine the programs they create. Such an experiment was performed in order to study other language features such as scope rules and the order of evaluation of expressions [2]. As a by-product of this work, we observed evidence of the effects of the various methods of associating a type with an operand [3]. This evidence indicated that the use of dynamically typed operands resulted in errors that remain in programs longer than the errors attributable to statically typed operands. Furthermore, programmers trying to convert an operand from one type to another, who were forced to grapple with the representation of an operand, committed errors that cast doubt upon their ability to write reliably in a language which treats its operands as collections of bits.

The compelling arguments concerning the superiority of statically typed languages and our own observations in previous work led to the design of an experiment to compare the effects of statically typed and "typeless" languages on programming reliability. Furthermore, if a statically typed programming language was superior, we hoped to be able to determine whether power or redundancy was the principal factor in the superiority.

2. Methodology

2.1. Overview

Two groups containing a total of 38 graduate and upper-level undergraduate students in a course at the University of Maryland programmed solutions to the same problem twice. One group of subjects programmed solutions first using a statically typed language and then using a "typeless" language. The second group also programmed solutions in the two languages, but used the languages in the opposite order. Each of the languages contained a substantial number of features common to other more widely used languages. The solutions to the relatively simple problem ranged from 48 to 297 lines. When a problem was assigned, each student was given the appropriate language manual containing similar sample programs. In order to avoid biasing the subjects, we did not devote class meetings to discussions of the features of the languages. However, we were available to the subjects on an individual basis. The subjects were cautioned to treat the problem as a take-home examination and to avoid discussing it with each other.

For the purposes of this study, a language was judged to enhance the reliability of software if the errors committed by its users were less frequent and severe than the errors committed by the users of a second language. The underlying hypothesis was that the errors that persist in the debugging process are similar to, but more numerous than, those that remain in completed programs.

This experiment represents an improvement over our previous work, since this experiment dealt with only a single language feature and each of the subjects used both languages. The first methodological change meant that differences in the frequency and severity of errors could now be attributed to a single language feature rather than to one of several altered features. More important, having each subject use both languages meant that we no longer relied on having groups of subjects with equal abilities; we could compare each subject's performance in one language to his own performance in the other language. However, having the subjects solve the same problem twice results in subjects' "learning solutions" to the problem on their first attempts.

2.2. The Languages

The first step in the methodology was to design the two languages to be used in the experiment. We attempted to make both languages identical in all features not affected by the issue of data types (e.g. control structures, scope rules, and declaration requirements). However, as data types and operations are intimately related, many of the operations in the languages have been altered. In addition, we intended that the features of both languages be shared with other widely used languages so that the results of the experi-

Table I. Comparison of Language Features.

Feature	ST Statically typed	NT Typeless
Types	Integers Fixed length strings	Words
Constants	Integers Strings	Integers Left-adjusted, blank-filled characters Right-adjusted, zero-filled characters Binary, octal, and hexadecimal numbers
Structuring	Arrays	Groups of words
Subscripting	Arrays only	Any word
Substring	Single character of string only	Any part of word
Other operations	No concatenate or search for strings No bit operations for integers	Bit operations: and, or, not
Input Stream Record	 Any constant Single strings (80 characters)	 Any constant 20 consecutive words (4 characters/word)
Output Stream Record	 Any expression Strings	 Any expression Pairs of addresses and lengths
Conversions	None	None

ment would be easy to interpret.

The "typeless" language (NT) has only single word variables. As in BCPL, programmers may associate an identifier with either a single word or a group of words. However, any identifier may be subscripted without error. In keeping with the idea of a "typeless" language, this decision allows the subscript operation to be applied to any identifier. A constant may have one of several representations, but its value must be storable in a single word. The numeric representations are binary, octal, decimal, and hexadecimal. The two character constant representations provided are those of BLISS: left-adjusted, blank-filled and right-adjusted, zero-filled. In order to manipulate the representations of data, a partword operation may be applied to select or alter an arbitrary portion of a word (e.g. a character). In addition to the common infix arithmetic operators, there are three infix bit operators: AND, OR, and NOT. The operators of NT can be applied to any operands of appropriate size. Stream input allows the constants of the language to appear as inputs, and stream output produces decimal representations of values. Record input reads an input record (i.e. card) into twenty consecutive words (four characters per word) starting at the location specified by the actual parameter. Record output accepts pairs of arguments X,Y and prints the character representation of the Y words starting at location X.

The statically typed language (ST) has integers and strings with arrays of data of both types. Strings are fixed length and are padded with blanks on the right in an assignment or input operation in which the value being given to the string is shorter than the declared length of the string. Besides assignment, comparison, input, and output, the only other string operation is the substring operation, which is restricted to single characters within a string. There are no concatenation or search operations for strings in the language, although these operations may be built by using substring and comparison. Integers have the common arithmetic operations, but no bit operations. Only identifiers declared to be arrays may be subscripted. The constants of ST are decimal numbers and strings. Stream and record input are restricted to scalar variables. (For example, the programmer must write a loop if he wishes to read or write an entire array of strings.) Stream input allows the constants of the language to appear as inputs, and stream output produces the appropriate representation of a value. Record input and output deal only with scalar strings, and values are padded on the right with blanks where appropriate.

It would hardly seem fair to provide subjects working in ST with all the operations normally associated with string processing (e.g. concatenate, search, substring, length, blank trimming, and conversion) and to require that the subjects working in NT build these

operations out of more primitive operations. If ST offered extra powerful operations in addition to its power and redundancy, it would almost certainly aid programming reliability more than NT. Instead, we restricted the string operations in the statically typed language to the essential string primitives: assignment, comparison, and selection of single characters within a string. This is an attempt to give the two languages "equally powerful" operations. Thus subjects in each language have to build the more powerful operations from sets of roughly comparable primitives. These differences are summarized in Table I.

Although it is obvious that both languages contain features common to many widely used languages, the differences between existing languages prevent our simply picking two of them to use in this study. Conversely, implementing the two compilers from scratch was an unappealing amount of work. Therefore we altered an existing language and its compiler, SIMPL-T [1], to produce both compilers. The SIMPL family is a set of programming languages under development at the University of Maryland. The term "family" implies that the languages contain some common basic features, such as data types and control structures. Each of the languages in the SIMPL family is built as an extension to a base language. Present members of the SIMPL family are the base "typeless" language SIMPL-X, the statically typed (integers and strings) language SIMPL-T, the mathematically oriented (reals) language SIMPL-R, and a systems implementation language for the PDP-11 SIMPL-XI.

The basic features of SIMPL-T that became part of both the ST and NT languages are:

1. A program consists of a sequence of procedures which can access a set of global variables, parameters, and local variables.
2. The statements are the assignment, if, while, case, call, return, exit, and abort. There is no block structure.
3. Procedures and functions may be recursive.
4. Procedures may not be passed as parameters. The default parameter passing mechanism is call by value, but call by reference may be specified.
5. The index of the first value of an array is zero.
6. Both stream and record input/output are available.

In addition, the features of NT and ST were added.

2.3. Data Collection

In order to reduce the cost of this experiment, each subject was asked to submit his intermediate listings to the experimenter. As a safeguard against uncooperative subjects, a data collection mechanism was built to automatically copy each program submitted by each subject. In order that this mechanism be inexpensive, the input cards of the subjects, rather than the output of the language processors, was copied. This allowed us to avoid copying the outputs of jobs that entered indefinite loops containing output statements and to rerun the jobs to insert diagnostic aids and to collect extra statistics.

Table II. Sample Input and Output for Problem.

Input	Output
01HGU 03SYSTEM	01HGU UGH 03SYSTEM METSYS METSYS METSYS

2.4. Identifying Errors

We examined the listings for errors. An error was discovered in one of three ways: by the appearance of a diagnostic message or incorrect output, by marks in the listing that subjects made during desk checking or debugging, or by us during our examination of the subjects' solutions to the problems.

In runs following the run on which an error appeared, a subject may have changed his program, either correcting the error or altering the manner in which the error manifested itself. Whether or not any manifestations of the error were evident on the subsequent runs, the error was said to persist if it remained uncorrected.

To determine the number of occurrences of the errors, the errors have been traced back to their origin and counted on all intervening runs. Errors that occurred in the source code have been traced back to the run in which the compiler first analyzed the source code. Errors that occurred in the input data have been traced back to the run in which the program first reached execution of a read statement.

2.5. Measures

The errors and occurrences described above are labeled total errors and total occurrences in the following sections. Errors and occurrences have been divided into two kinds, primary and equivalent, which together comprise total errors and total occurrences. The following example illustrates the differences between primary and equivalent errors. Suppose a subject changed the number of formal parameters accepted by a procedure, but changed only $N - M$ of the N procedure invocations to have the correct number of actual parameters. Clearly the M procedure invocations with the incorrect number of actual parameters are errors, but they are not as serious as M distinct errors. However, each of the M incorrect invocations could stop the program from executing correctly. Therefore one of the errors (the one that occurred on the most consecutive runs) has been designated as the primary error, and the other $M - 1$ errors have been called equivalent errors. The tables that follow present primary and total (i.e. primary plus equivalent) errors and occurrences.

In addition, the number of runs that contained any errors has been calculated. This measure, called error

runs, is an attempt to measure the severity of errors rather than just the gross number of errors. A single error that occurred on five consecutive runs has error runs equal to five, while five errors that occurred on the same run each have error runs equal to one.

2.6. The Problem

Designing a problem which is typical of the entire range of programming applications is an impossible task. Instead we tried to design a task that required the use of the altered features in the two languages (i.e. string assignment, substring assignment and selection, and conversion from character to integer representations). Subjects were asked to read a series of cards of the form *ddcc . . . c* where *d* is a digit between 0 and 9 and *c* is a character and to produce *dd* copies of the reversed character string. Each subject's program was to read the card, place it in an array to be printed, process the card (i.e. produce *dd* copies of the reversed characters), place the reversed strings in the output array, and print the contents of the array. The program was also to contain a recursive procedure to reverse the character portion of the card. Sample input and output are shown in Table II (p. 587).

While the problem is not difficult, it is not trivial either. The solutions ranged from 48 to 216 lines and 27 to 85 statements in ST, and from 58 to 297 lines and 31 to 105 statements in NT. Although this is not a large program, much of the experimental work being done by most researchers concentrates on much smaller programs (e.g. ten statements). However, should we work with programs that become too large, it may become difficult to determine if any errors remain in them.

Since the problem was assigned as a normal homework assignment for the course, subjects were well motivated to complete the problem. The subjects were given two weeks to solve the problem. Sample solutions to the problems may be found in Appendix B (see pp. 594–595).

2.7. Statistical Techniques

The averages per programmer for the number of errors, total errors, occurrences, total occurrences, and error runs have been calculated for a language for each hypothesis. Because averages over small samples can be greatly distorted by one or two very bad performances, nonparametric tests have been used to determine the statistical significance of all the differences.

Nonparametric tests were used because they are not related specifically to the parameters of a given population and may therefore be used under very general conditions (for instance, if one of the sample populations displays considerably more variance than another). Nonparametric tests waste information because they compare the signs or ranks of the values of populations rather than the values themselves. This leads to a greater risk of accepting a false null hypothesis than the alternative standard tests.

Table III. First Solution.

Measure	NT	ST	Level
Errors	19.40	12.44	
Total Errors	26.87	22.39	
Occurrences	90.80	39.17	<5%
Total Occurrences	125.80	51.72	<5%
Error Runs	21.20	10.83	<5%

Table IV. Second Solution.

Measure	NT	ST	Level
Errors	13.17	7.87	<2%
Total Errors	24.00	8.93	<.2%
Occurrences	59.83	29.20	<5%
Total Occurrences	99.61	31.40	<1%
Error Runs	14.83	10.07	

Table V. NT1/ST2.

Measure	NT	ST	Level
Errors	19.40	7.87	<.5%
Total Errors	23.87	8.93	<2.5%
Occurrences	90.80	29.20	<.5%
Total Occurrences	125.80	31.40	<.5%
Error Runs	21.20	10.07	<2.5%

Table VI. ST1/NT2.

Measure	NT	ST	Level
Errors	13.17	12.44	
Total Errors	24.00	22.39	
Occurrences	59.83	39.17	<.5%
Total Occurrences	99.61	51.72	<1%
Error Runs	14.83	10.83	<1%

Four nonparametric tests were used in evaluating the results of this experiment: the Mann-Whitney U-test, the Wilcoxon matched-pairs signed-ranks test, the Spearman rank correlation coefficient, and the Kruskal-Wallis one-way analysis of variance [7]. A two-tailed Mann-Whitney test was applied to compare the performances of the two independent groups (the NT and ST groups) on their first and second solutions to the problem. A two-tailed test was chosen despite our belief that, given two groups of subjects of compatible ability, the ST group would perform better than the NT group. We felt there was a possibility that one of the groups might contain superior programmers. In order to compare the performances of the same subjects in each of the languages (two related sets of measures), a

one-tailed Wilcoxon test was used. Here the one-tailed test was appropriate because we believed that any one subject would commit fewer errors solving the problem in ST than in NT. In addition, we felt that certain subjects would benefit more than others from using ST as opposed to using NT. Two tests, the Spearman and Kruskal-Wallis tests, were used to investigate the size of the improvements in the error measures attributable to the subjects' use of ST. To accomplish this, a subject's ST error measures were subtracted from his NT error measures. The subjects were then ranked in ascending order. The subjects with lower NT than ST error measures were assigned the lower ranks, followed by the subjects who had increasingly lower ST than NT measures. The Spearman test was used to measure the correlation between higher examination scores and smaller improvements in the ST error measures. The Kruskal-Wallis test was used to determine if any of three groupings of subjects based on experience exhibited different improvements in error measures using ST.

3. Results

3.1. Summary

Fifteen of the original 17 subjects who wrote their first solutions in NT and 18 of the original 21 subjects who wrote their first solutions in ST finished solutions in each of the languages. A total of 4834 occurrences of 1372 errors were committed on 1014 runs by these 33 subjects during the experiment. Using ST, the subjects made an average of 11.61 runs and had an average of .21 errors remaining in their final solutions. Using NT, the comparable figures were 19.12 runs and .64 errors remaining in the final runs. Tables III–VI contain the averages per subject for the number of errors, total errors, occurrences, total occurrences, and error runs. In the first solutions to the problem, the subjects using ST had lower error measures than the subjects using NT; however, only the differences in occurrences, total occurrences, and error runs are statistically significant (Mann-Whitney U-test, two-tailed). The total errors show a 13:11 advantage for the ST users, while the total occurrences exhibit a 5:2 advantage. Thus we might conclude that the use of ST does not reduce the original commission of errors so much as it detects them quickly.

When the subjects solved the problem a second time using a different language (i.e. the subjects who used NT in the first solutions now used ST and vice versa), all the measures again favored the subjects using ST. As was the case with the first solutions, the ST advantage for total occurrences (10:3) outweighed that for total errors (8:3).

Subjects in the group that used NT first and then ST (NT1/ST2) and those in the group that used ST first and then NT (ST1/NT2) both committed fewer and less severe errors using ST than using NT. However, while the differences in the measures are highly significant for the NT1/ST2 group (Wilcoxon matched-pairs signed-ranks test, one-tailed), the differences in errors and total errors are not statistically significantly different for the ST1/NT2 group.

There are at least two effects to consider: the familiarity of the subjects with the problem, and the language being used to solve the problem. In solving the problem the second time, the NT1/ST2 subjects benefited both from the knowledge they gained solving the problem the first time and from using the data type features of ST. The knowledge gained during the first solution was both problem-related (e.g. how to use recursion and when to stop it) and language-related (e.g. the scope rules and control features common to both languages). In contrast, the subjects in the ST1/NT2 group benefited mainly from familiarity with the problem. Another factor which may help explain the lack of significance of the error and total error results for the ST1/NT2 group is that some of these subjects

Table VII. Character Selection: Position/Length Errors.

Measure	Number
Errors	16
Total Errors	50
Occurrences	139
Total Occurrences	343
Error Runs	120
Subjects Involved	10

Table VIII. Word Selection: Position/Length Errors.

Measure	Number
Errors	22
Total Errors	72
Occurrences	203
Total Occurrences	246
Error Runs	170
Subjects Involved	15

Table IX. Constants Too Long.

Measure	Number
Errors	23
Total Errors	54
Occurrences	42
Total Occurrences	89
Error Runs	43
Subjects Involved	17

used the types and operations of ST (e.g. characters and substrings) as abstract operations in solving the problem by using NT. (See the sample NT solution in Appendix B.) A complete breakdown of the measures for each subject may be found in Appendix A (see pp. 592–593).

3.2. Detailed Analysis of Errors

3.2.1. NT errors. Subjects working in NT had more difficulty coping with the representation of string data (i.e. its origin, length, and justification) than with differences between types of data (i.e. integer and character format). In selecting characters within words, subjects either used 1 (instead of 0) as the index of the leftmost bit within a word and 9 as the length of a character in bits (character positions 1, 10, 19, 28), or 1 as both the index of the leftmost bit and the length of a character (character positions 1, 2, 3, 4), or 35 as the index of the leftmost bit and 9 as the length of a character (character positions 35, 27, 18, 9). Table VII contains a summary of these kinds of errors made by the 10 NT subjects who made them (i.e. Subjects Involved).

NT subjects made similar errors dealing with groups of words used to represent strings of characters (Table VIII). Errors arose from using 1 (instead of 0) as the index of the first word in a group of words and from using either 1 or 80 (instead of 20) as the length of a group of words used to represent a string of characters.

In addition to these selection errors, subjects using NT had to cope with considerably more complex selection mechanisms. In ST, characters are selected via a substring operation which requires a single argument, the position of the character in the string. In NT, characters are selected by using both a subscript operation, which requires the location of the word in which the character occurs, and a partword operation, which requires the location of the bit at which the character occurs and the length of the character in bits. Thus, in order to process strings of characters, a subject must maintain one index in ST but two indices in NT. Similar problems arise in processing a group of words as an array of character strings. The subscript operation in ST is applied to an array of strings to select a single string; the subscript operation in NT must be used to select both a particular string and a word containing a character in that string. Some subjects working in NT actually broke up the subscript computation into a string selector and a word selector but occasionally forgot to include one of these components while processing a string. Other NT subjects combined the calculation and miscalculated the boundaries of strings.

Subjects using NT also had difficulty with the length of constants (Table IX) and their justification and fill (Table X). Character and octal constants, which were restricted to values which could be placed in a single word, were often too long.

Character constants, especially single character blanks and digits, were often incorrectly justified and filled. The large discrepancy in the measures for the two types of justification errors can probably be attributed to the backgrounds of the subjects. Left-justified

Table X. Incorrect Justification of Constants.

Measure	Left-justified	Right-justified
Errors	21	1
Total Errors	79	1
Occurrences	223	1
Total Occurrences	638	1
Error Runs	185	1
Subjects Involved	17	1

Table XI. Subscripting Formal Parameters.

Measure	Number
Errors	7
Total Errors	10
Occurrences	61
Total Occurrences	79
Error Runs	55
Subjects Involved	6

Table XII. Incompatible Types.

Measure	Operators	Intrinsics
Errors	3	23
Total Errors	4	52
Occurrences	9	99
Total Occurrences	11	174
Error Runs	9	98
Subjects Involved	3	19

Table XIII. Representation of Arrays/Strings.

Measure	Number
Errors	4
Total Errors	20
Occurrences	21
Total Occurrences	61
Error Runs	22
Subjects Involved	4

Table XIV. Origins of Arrays/Strings.

Measure	Strings	Arrays
Errors	4	2
Total Errors	4	2
Occurrences	15	3
Total Occurrences	15	3
Error Runs	13	3
Subjects Involved	3	2

Table XV. Representation of Constants.

Measure	Number
Errors	5
Total Errors	5
Occurrences	16
Total Occurrences	16
Error Runs	16
Subjects Involved	4

Table XVI. Incompatible Types.

Measure	Operator	Intrinsic
Errors	9	7
Total Errors	29	7
Occurrences	37	24
Total Occurrences	70	24
Error Runs	37	24
Subjects Involved	9	7

Table XVII. Substring Assignment to Uninitialized Variables.

Measure	Number
Errors	19
Total Errors	19
Occurrences	125
Total Occurrences	125
Error Runs	106
Subjects Involved	15

blank-filled character constants are specified (as are string constants in most programming languages) by enclosing the character in single quotation marks. Right-justified zero-filled character constants are specified in double quotation marks. The NT convention was not designed to mislead the subjects intentionally but follows the conventions established by other type-less languages (e.g. BLISS). The partword operation in NT returns a right-justified zero-filled value which subjects often tried to compare to a left-justified blank-filled constant. There was only one error which involved a subject working with variables having different justifications.

NT subjects also had trouble with the parameter passing mechanism, which defaulted to call by value in each of the languages. In order to apply the subscript operation correctly to a formal parameter in NT, the parameter had to be passed by reference. (Otherwise a temporary location is subscripted.) Six NT subjects committed errors of this kind (Table XI).

In contrast to the problems subjects had with the representation of data in NT, relatively few errors resulted from uses of data of the wrong type. One of these errors resulted in incrementing the elements of an array instead of shifting them from right to left. That is,

Card(Word) := Card(Word) + 1

instead of

Card(Word) := Card(Word + 1)

Only in using the NT intrinsics to read and write values (READ/WRITE for integers and READC/WRITEL for characters) did the subjects make a large number of errors (Table XII).

3.2.2. ST errors. Subjects using ST had relatively little trouble with either the representation or the type of data. Several subjects did attempt to treat strings of length 80 as arrays of 80 strings of length 1 and vice versa (Table XIII).

There were also several errors caused either by using zero (instead of one) as the index of the first character in the string or by using one (instead of zero) as the index of the first element of an array (Table XIV). Errors involving the incorrect representation of constants were also rare (Table XV).

Examining operations on data of an incorrect type, we find that ST subjects made a few more errors involving operators and a few less errors involving I/O intrinsics than did the same subjects using NT (Table XVI).

The most frequent errors made by the subjects using ST were attempts to use the substring operation to assign a value to a variable which had not been initialized (Table XVII).

3.3. Subjects Aided by ST

Examining the tables that summarize the performances of the subjects in Appendix B, we can easily see that, although most subjects benefited from using ST, not all of the them did. Furthermore, even among those subjects whose performances improved, the amount of improvement differed radically. This made us wonder

Table XVIII. High Scores and Less Improvement Using ST.

Measure	Correlation Coefficient	Level
Errors	.18	
Total Errors	.42	<2%
Occurrences	.19	
Total Occurrences	.37	<5%
Error Runs	.21	

Table XIX. Differences between NT and ST Error Measures Within Groups of Subjects Based on Experience.

Measure	0	1	More	Level
Errors	8.50	4.55	-3.17	
Total Errors	21.10	10.45	-1.67	<20%
Occurrences	62.50	39.91	21.17	
Total Occurrences	134.50	58.36	25.08	<10%
Error Runs	8.40	11.00	2.93	

which (if any) group of subjects benefited most from using ST.

The first comparison we made involved examination scores. We felt that better students might be more disciplined programmers and either benefit less from the abstractions offered by the data types of ST or cope better with the representation of data in NT. To examine this hypothesis, we ranked the subjects twice: first in descending order according to their average examination scores, and then in ascending order according to the differences between the error measures for their NT and ST performances. Those subjects at the head of the second ranking benefited least from using ST. They had negative differences between the NT and the ST measures, having more errors, occurrences, and error runs in ST than in NT. A Spearman rank correlation coefficient showed two significant positive correlations between subjects with higher examination averages and those with smaller improvements in error measures using ST (Table XVIII).

The final comparison we made concerned the effect of a subject's experience on his performance. We measured experience in terms of the number of languages in which the subject had written 25 or more programs. As programmers learn new languages, each new language requires less learning investment than the previous one. Thus we felt the more languages with which a subject was familiar, the easier time he would have learning both new languages. We broke the subjects into three groups based on the number of languages that they indicated they had used frequently (10 subjects with zero languages, 11 subjects with one lan-

Table XX. NT1/ST2.

Subject	Errors	Total Errors	Occur	Total Occ	Error Runs
1	11/21	16/21	50/61	82/61	16/22
2	1/4	1/5	1/5	1/6	1/4
3	17/7	42/7	121/27	200/27	18/10
4	85/11	126/12	365/29	491/30	68/8
5	7/2	7/2	38/3	38/3	9/2
6	16/5	30/5	197/18	240/15	70/12
7	23/3	40/3	84/10	217/10	16/5
8	5/2	10/5	10/3	23/6	5/2
9	9/9	12/9	10/42	13/42	5/13
10	13/2	17/2	53/4	81/4	13/3
11	26/24	30/32	101/136	136/159	19/25
12	10/3	10/3	46/14	46/14	11/8
13	11/12	11/14	115/24	115/26	26/20
14	19/7	24/8	95/53	119/56	20/13
15	21/6	27/6	76/9	85/9	21/4

Table XXI. ST1/NT2.

Subject	Errors	Total Errors	Occur	Total Occ	Error Runs
16	2/7	2/36	2/20	2/58	1/8
17	5/9	5/9	13/19	13/19	4/4
18	27/16	48/22	93/90	131/111	18/16
19	21/15	41/17	152/171	177/174	32/39
20	3/10	3/23	5/19	5/50	3/5
21	21/13	24/50	45/69	48/248	9/12
22	1/17	20/7	19/18	40/18	7/10
23	11/20	14/33	25/137	30/190	11/32
24	9/19	15/34	28/57	36/82	9/17
25	10/8	21/10	28/25	48/28	10/6
26	36/26	40/50	127/156	136/382	38/38
27	12/14	85/26	31/63	111/92	11/12
28	7/8	16/12	10/19	19/38	5/7
29	8/7	11/7	15/13	18/13	7/4
30	4/5	4/14	11/20	11/65	5/7
31	22/34	32/50	59/123	73/152	17/26
32	7/12	12/20	13/47	21/66	3/19
33	9/7	10/12	11/11	12/17	5/5

guage, and 12 subjects with more than one language). The figures in Table XIX again represent the differences between ST and NT performances. For example, subjects who were not experienced with any other programming language made 8.50 more errors in NT than ST. It appears that the improvement in performance from NT to ST varies inversely with experience as measured by the number of frequently using programming languages.

4. Conclusions and Future Work

This experiment shows that at least in our environment, the features of a statically typed language increase programming reliability more than the features of a "typeless" language. Of course these results come as no surprise. The statically typed language still has better primitives with which to solve string-processing problems. In addition, most students learn a statically typed language as their first language and make errors when encountering a new concept like a "typeless" language for the first time.

Furthermore, the detailed analysis of the magnitude and kinds of errors committed by the subjects seems to indicate that the power of the statically typed language aided the subjects more than the redundancy did. Finally, subjects who are less able and experienced are helped most by a statically typed language.

There is no reason to assume that languages designed without explicit concern for reliability will be suitable for the production of reliable software. Both the "typeless" and statically typed languages share features with other widely used programming languages. The empirical evidence gathered in this research should help in selecting an existing language in which to implement a piece of software and serve as an objective basis in the design of new programming languages.

There are several possible directions for future work. Additional experimental work with data types could involve comparing the statically typed language to another statically typed language with extended string operations (e.g. concatenation, multiple character substring, conversions, etc.) to discover the possible advantages of providing these high-level operations. The statically typed language could also be compared to an identical language whose compiler/interpreter waited until run time to detect type mismatches. This experiment would point out the advantages of compile-time error checking. The experimental approach could also be applied to other language features as well as to measuring programmer aptitude or productivity. Language designers are continually proclaiming constructs "harmful" and proposing alternative features. We are anxious to perform experiments on some of these features (e.g. interfaces, assignment statements, synchronization primitives, etc.) to determine if the alternative features achieve their goals.

Acknowledgments. We have benefited from discussions on this topic with J.J. Horning of the University of Toronto, M.V. Zelkowitz, R.G. Hamlet, V.R. Basili, and B. Shneiderman of the University of Maryland, R.E. Noonan of the College of William and Mary, and the members of IFIP Working Group 2.4 (Machine-Oriented Higher-Level Languages). A.J. Turner of Clemson University provided valuable information about the SIMPL-T compiler. We are also grateful to the referees for their helpful comments on an earlier version of this paper.

References

1. Basili, V.R., and Turner, A.J. A transportable extendable compiler. *Software—Practice and Experience 5* (1975), 269-278.
2. Gannon, J.D., and Horning, J.J. Language design for programming reliability. *IEEE Trans. Software Eng. SE-1,* 2 (June 1975), 179-191.
3. Gannon, J.D. Data types and programming reliability: some preliminary evidence. MRI Symp. on Comptr. Software Eng., Vol. 24, Polytechic Press, Polytechnic Institute of N. Y. (1976), 367-376.
4. Liskov, B.H., and Zilles, S.N. Programming with abstract data types. SIGPLAN Notices (ACM) *9,* 4 (April 1974), 50-59.
5. Reynolds, J.C. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM 13,* 5 (May 1970), 308-319.
6. Richards, M. BCPL: a tool for compiler and system writing. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 557-566.
7. Siegel, S. *Nonparametric Statistics for the Behavioral Sciences.* McGraw-Hill, New York, 1956.
8. Wirth, N. The programming language Pascal. *Acta Informatica 1,* 1 (1971), 35-63.
9. Wulf, W.A., Russell, D.B., and Habermann, A.N. BLISS: a language for systems programming. *Comm ACM 14,* 12 (Dec. 1971), 780-790.
10. Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Eng. SE-2,* 4 (Dec. 1976), 253-264.

Appendix A

Tables XX and XXI contain a breakdown of the measures for each subject by language. Each column contains two figures separated by a slash (/). The first number is a measure of the subject's performance in the first language he used and the second number is the comparable measure in the second language.

Appendix B

The following two programs are sample solutions to the problem described in Section 2.6. The first solution is written in ST and the second solution is written in NT.

(Appendix B continues on next page)

ST Sample Solution

```
/* READ IN STRINGS OF THE FORM DDCC...C AND PRODUCE */
/* DD COPIES OF THE REVERSED STRING C...CC          */

STRING ARRAY OUTPUT[80](20) = (' '(20))
INT FIRSTCHARPOS = 1,
    LASTCHARPOS = 80,
    FIRSTSTRPOS = 0,
    STRINGSIZE = 1,
    NUMDIGITS = 2      /* NUMBER OF DIGITS TO CONVERT */
STRING BLANK[1] = ' '

INT FUNC NUM(STRING CVAL)

/* RETURN NUMERIC EQUIVALENT OF FIRST CHARACTER OF CVAL */

  CASE CVAL[1] OF
      \'0'\ RETURN(0)
      \'1'\ RETURN(1)
      \'2'\ RETURN(2)
      \'3'\ RETURN(3)
      \'4'\ RETURN(4)
      \'5'\ RETURN(5)
      \'6'\ RETURN(6)
      \'7'\ RETURN(7)
      \'8'\ RETURN(8)
      \'9'\ RETURN(9)
      ELSE
         WRITEL('ILLEGAL CHARACTER CONVERSION')
         RETURN(0)
      END


PROC PRODUCECOPIES(STRING IN,REF INT LOC)

/* PLACE ONE COPY OF THE STRING AND
   DD COPIES OF THE REVERSED STRING IN 'IN' */

  INT I,
      FIRST, /* START OF STRING TO BE REVERSED */
      LAST,  /* END OF STRING TO BE REVERSED */
      COPIES /* NUMBER OF REVERSED COPIES TO BE PRODUCED */

  FIRST := FIRSTCHARPOS
  COPIES := NUM(IN[FIRST]) * 10 + NUM(IN[FIRST+1])
  FIRST := FIRST + NUMDIGITS
  LAST := LASTCHARPOS

  WHILE LAST >= FIRST .AND. IN[LAST] = BLANK
     DO /* LAST POINTS TO FINAL NONBLANK CHARACTER */
     LAST := LAST - 1
     END /* LAST POINTS TO LAST NONBLANK CHARACTER */

  I := FIRSTCHARPOS
  WHILE I <= LAST
     DO /* COPY UNREVERSED STRING TO OUTPUT ARRAY */
     OUTPUT(LOC)[I] := IN[I]
     I := I + 1
     END
  LOC := LOC + STRINGSIZE /* NEXT AVAIL OUTPUT POS */

  IF COPIES > 0
     THEN /* REVERSE STRING AND MAKE COPIES */
        CALL REV(OUTPUT(LOC),FIRSTCHARPOS,OUTPUT(LOC-STRINGSIZE),
             FIRST,LAST)
        LOC := LOC + STRINGSIZE
        COPIES := COPIES - 1
        WHILE COPIES > 0
           DO /* MAKE ADDITIONAL COPIES OF REVERSED STRING */
           OUTPUT(LOC) := OUTPUT(LOC-STRINGSIZE)
           LOC := LOC + STRINGSIZE
           COPIES := COPIES - 1
           END
     END

REC PROC REV(REF STRING OUT,INT OUTPOS,STRING IN,INT FIRST,
             INT LAST)

/* 'LAST' CHARACTER OF 'IN' TO 'FIRST' POSITON OF 'OUT' */

  IF LAST < FIRST
     THEN
        RETURN
     ELSE
        OUT[OUTPOS] := IN[LAST]
        CALL REV(OUT,OUTPOS+1,IN,FIRST,LAST-1)
     END


PROC DRIVER

  STRING CARD[80]
  INT LASTRECORD,I

  LASTRECORD := FIRSTSTRPOS
  WHILE .NOT. EOIC
     DO /* READ AND REVERSE INPUT */
     READC(CARD)
     CALL PRODUCECOPIES(CARD,LASTRECORD)
     END

  I := FIRSTSTRPOS
  WHILE I <= LASTRECORD-STRINGSIZE
     DO /* PRINT OUTPUT ARRAY CONTENTS */
     WRITEL(OUTPUT(I))
     I := I + STRINGSIZE
     END

START DRIVER
```

NT Sample Solution

```
/* READ IN STRINGS OF THE FORM DDCC...C AND PRODUCE */
/* DD COPIES OF THE REVERSED STRING C...CC          */
VAR OUTPUT(300) = (''(300))
VAR FIRSTCHARPOS = 0,
    LASTCHARPOS = 79,
    FIRSTSTRPOS = 0,
    STRINGSIZE = 20,
    CHARLEN = 9,
    CHARPERWD = 4,
    NUMDIGITS = 2,      /* NUMBER OF DIGITS TO CONVERT */
    BLANK = " "
FUNC WD(VAR POS)
/* RETURN THE WORD INDEX CONTAINING THE POS-TH CHARACTER */
  RETURN(POS/CHARPERWD)
FUNC CHAR(VAR POS)
/* RETURN THE BIT INDEX OF THE POS-TH CHARACTER */
  RETURN((POS-POS/CHARPERWD*CHARPERWD)*CHARLEN)

FUNC SUBSTR(REF VAR STR,VAR POS)
/* RETURN THE POS-TH CHARACTER OF STR */
  RETURN(STR(WD(POS))[CHAR(POS),CHARLEN])
REC PROC REV(REF VAR OUT,VAR POS,REF VAR IN,VAR FIRST,VAR LAST)
/* 'LAST' CHARACTER OF 'IN' TO 'FIRST' POSITON OF 'OUT' */
  IF LAST < FIRST
     THEN
        RETURN
     ELSE
        OUT(WD(POS))[CHAR(POS),CHARLEN] := SUBSTR(IN,LAST)
        CALL REV(OUT,POS+1,IN,FIRST,LAST-1)
     END
PROC PRODUCECOPIES(REF VAR IN,REF VAR LOC)
/* PLACE ONE COPY OF THE STRING AND
  DD COPIES OF THE REVERSED STRING IN 'IN' */
  VAR I,
      DIGIT,
      FIRST, /* START OF STRING TO BE REVERSED */
      LAST,  /* END OF STRING TO BE REVERSED */
      COPIES /* NUMBER OF REVERSED COPIES TO BE PRODUCED */
  FIRST := FIRSTCHARPOS
  COPIES := 0
  I := FIRST
  WHILE I <= FIRST + 1
     DO /* CONVERT REPETITION COUNT TO INTEGER */
     DIGIT := SUBSTR(IN,I) - "0"
     IF DIGIT < 0 .OR. DIGIT > 9
        THEN /* ILLEGAL DIGIT */
           WRITEL('ERR',1)
           DIGIT := 0
        END
     COPIES := COPIES * 10 + DIGIT
     I := I + 1
     END
  FIRST := FIRST + NUMDIGITS
  LAST := LASTCHARPOS
  WHILE LAST >= FIRST .AND. SUBSTR(IN,LAST) = BLANK
     DO /* LAST WILL POINT TO FINAL NONBLANK CHARACTER */
     LAST := LAST - 1
     END
  I := FIRSTCHARPOS
  WHILE I <= LAST
     DO /* COPY UNREVERSED INPUT TO OUTPUT ARRAY */
     OUTPUT(LOC+WD(I))[CHAR(I),CHARLEN] := SUBSTR(IN,I)
     I := I + 1
     END
  LOC := LOC + STRINGSIZE /* NEXT AVAIL OUTPUT POS */
  IF COPIES > 0
     THEN /* REVERSE STRING AND MAKE COPIES */
        CALL REV(OUTPUT(LOC),FIRSTCHARPOS,OUTPUT(LOC-STRINGSIZE),FIRST,LAST)
        LOC := LOC + STRINGSIZE
        COPIES := COPIES - 1
        WHILE COPIES > 0
           DO /* MAKE ADDITIONAL COPIES OF REVERSED STRING */
           I := 0
           WHILE I <= LAST
              DO /* COPY REVERSED STRING TO OUTPUT ARRAY */
              OUTPUT(LOC+WD(I))[CHAR(I),CHARLEN] :=
                 SUBSTR(OUTPUT(LOC-STRINGSIZE),I)
              I := I + 1
              END
           LOC := LOC + STRINGSIZE
           COPIES := COPIES - 1
           END
     END
PROC DRIVER
  VAR CARD(20)
  VAR LASTRECORD,I
  LASTRECORD := FIRSTSTRPOS
  WHILE .NOT. EOIC
     DO /* READ AND REVERSE INPUT */
     READC(CARD)
     CALL PRODUCECOPIES(CARD,LASTRECORD)
     END
  I := FIRSTSTRPOS
  WHILE I <= LASTRECORD - STRINGSIZE
     DO /* PRINT OUTPUT ARRAY CONTENTS */
     WRITEL(OUTPUT(I),STRINGSIZE)
     I := I + STRINGSIZE
     END
START DRIVER
```