Comparing programming paradigms: an evaluation of functional and object-oriented programs

by R. Harrison, L.G. Samaraweera, M.R. Dobie and P.H. Lewis

A quantitative evaluation of the functional and object-oriented paradigms is presented. The aim of the project is to investigate whether the quality of code produced using a functional language is significantly different from that produced using an object-oriented language. Twelve sets of algorithms are developed, together with a number of utility functions, in both SML and C++. Strict constraints are imposed during the development cycle to improve the reliability of the results. The statistical tests do not reveal any significant differences for direct measures of the development metrics used which are associated with quality, such as the number of known errors, the number of modification requests, a subjective complexity assessment etc. However, significant differences are found for an indirect measure, the number of known errors per thousand non-comment source lines, and for various code metrics, including the number of distinct functions called, the number of distinct library functions called, and the ratio of these, which is a measure of code reuse. A difference is also found for the time taken to test the programs, due to different compilation techniques and a difference in the number of test cases executed.

1 Introduction

This paper presents the results of a quantitative evaluation of the functional and object-oriented paradigms. The aim of the project is to investigate whether the quality of code produced using a functional language is significantly different from that produced using an object-oriented language.

Before undertaking this analysis, we performed a preliminary investigation to establish a base-line of quality indicators to determine whether certain design and code measures could be used as metrics for the remainder of the project. The results of these preparatory studies have been reported in detail [1, 2] and are discussed briefly in Section 2.

The aim of the research presented here is to answer the following question: does the quality of code produced using a functional language differ from that produced using an object-oriented language?

This paper reports on the results of performing a quantitative analysis of certain internal attributes which reflect various quality-type properties for a set of object-oriented and functional programs.

2 Measuring software quality

Defining and measuring software quality is extremely difficult. The classification, validation and application of metrics has aroused much debate. Fenton [3] proposes a classification system for software measurement which has three main headings for entities whose attributes may be measured: product, process and resource. In addition, the attributes of entities (such as size, modularity, reuse, etc.) may be either internal (meaning that they can be measured in terms of the product, process or resource itself) or external (in which case they can only be measured by considering their relationships with the environment).

In this project our aim is to quantify the quality of the delivered code. Of particular interest are the external product attributes of the code, such as reliability, usability, maintainability, testability, reusability etc. However, these external attributes are extremely difficult to quantify. Instead of trying to measure them directly, we performed an experiment in which two sets of measurements are collected during the software development process. The first set consists of a number of development metrics assumed to be indicative of the external quality-type attributes of the code [3, 4].

- the number of known errors found during testing (KE).
- the time to fix known errors (TKE).

- the number of modifications requested during code reviews, testing and maintenance (MR).
- the time to implement modifications (TMR).
- a subjective assessment of complexity, provided by the system developers (SC).

The second set consists of measures of certain internal attributes of the code, such as length, the number of functions called, the number of functions declared etc., which can be easily measured by examining the delivered code. These measures of internal attributes, which we refer to as suggested indicators or code metrics, are collated and tested for correlation against the development metrics listed above.

Using Fenton's classification system, the development metrics would be classified as internal process measures, whereas the code metrics would be classified as internal product measures.

Thus we address two questions within one project:

(*a*) is there any correlation between this set of development metrics (known errors, modification requests, subjective complexity etc.) and the set of chosen suggested indicators (non-comment source lines, number of distinct domain-specific functions called, number of function declarations etc.)?
(*b*) is there any significant difference between the quality of the code for the two different languages?

If a significant correlation is found between the development metrics and the suggested indicators in (*a*) above, then

(*a*) in future projects we could use the static analysis of delivered code to determine its quality as far as modules which are prone to faults, changes or subjective complexity are concerned; noting known errors and so on during testing then becomes unnecessary.
(*b*) existing code could be assessed for such features very simply.
(*c*) both the development metrics and the suggested indicators could be used to answer question (*b*) above.

Significant correlations were detected for many of the suggested indicators [1, 2]. In particular, the number of non-comment source lines was found to be closely correlated to the development metrics for both paradigms. In addition the number of distinct functions called, the number of domain-specific functions called and the number of function definitions were found to be correlated to the number of modification requests for both paradigms. Further discussion of this research is found in Section 6.

An answer to question (*b*) above will enable us to offer guidelines to software developers who face a choice of languages when implementing particular applications. If no significant difference is found, then other considerations, such as a subjective preference, may become important factors in the decision.

3 Method

During the project, 12 sets of algorithms were developed, together with a number of utility functions in both SML and C++. The algorithms were all from the image analysis domain. Strict constraints were imposed during the development cycle to reduce the confounding effects of the experimental variables, as described below. The project team adhered to industrial practices as much as possible, adopting quality assurance methods from industry for the production of requirements, design and testing documentation, as well as following standard techniques from industry for walkthroughs and code reviews.

3.1 Application domain

When choosing an application domain for the software development, we were seeking problems that would exercise the languages in a reasonably wide range of tasks. We required a domain that demanded a range of data types, good file handling facilities and significant input/output and user–interface facilities. It also had to provide a range of algorithms of differing complexities, and ideally needed to be a domain in which we already had significant software engineering expertise. This led us to image analysis, which is a fertile area for research in software engineering.

An increasing number of computer applications involve the manipulation and analysis of images and video sequences, in addition to the traditional processing of text and numerical data. The field of image analysis, which is still developing rapidly, covers a wide variety of different types of algorithm. These range from low-level image processing involving image to image transformations, through intermediate-level feature extraction, to high-level symbolic reasoning and knowledge-based systems at the image understanding/computer vision end of the field.

The algorithms used in image analysis are becoming increasingly complex, and hence are difficult to develop and implement. In addition, they benefit from being encoded in such a way that they can be combined and recombined in various experimental arrangements that eventually lead to efficient and practical image applications. Casting such algorithms in the framework of object-oriented and functional languages was expected to provide more robust, easily understood and maintainable software. Related research [5–7] indicated that this approach to the production of image processing algorithms would be productive. Restricting the application domain reduced the number of variables in the experiment, and so helped to produce more reliable results for comparison. A range of algorithms was specified, designed and implemented, varying from low-level algorithms (for example, convolution algorithms) to intermediate-level algorithms such as curvilinear feature extraction algorithms.

Although it is never possible to generalise from one application area to all applications, we believe that the algorithms developed here cover a sufficiently wide range of software engineering problems for the results to provide useful indicators for other application domains.

3.2 Reducing bias

The result of the investigation may be confounded by a number of variables:

☐ the experience of the developers with the application.

□ the experience of the developers with the programming languages.
□ the application domain.
□ the development environment.

To try to reduce the impact of these variables, we chose a developer who had comparable experience of programming with both SML and C++, and could program competently with both languages, but had used neither language within the chosen application domain before the start of the project.

The application domain, image analysis and the development environment were kept constant throughout, except for a C++ compiler upgrade which occurred halfway through the project. This necessitated one minor modification, which is included in the recorded statistics. The developer implemented a number of utility functions and nine image analysis algorithms in SML during the first three months of the project; during the second three months the same algorithms were implemented in C++. For the second half of the project, two more substantial image analysis algorithms were implemented in C++ and then in SML. It was hoped that switching the order of development in this way would help to reduce the confounding effects which may occur due to the developer's increased knowledge of the application domain and of the programming languages. The programs developed during the second half of the project were fairly complicated and relied on some of the software developed earlier. Consequently, it was necessary to maintain the programs developed early in the project life-cycle for the duration of the project.

The developer was assigned full time to the project, and worked in conjunction with application domain experts who acted as customers by specifying requirements and providing acceptance test data. The expected test output was produced, wherever possible, by testing the developed programs against existing programs written in imperative languages (back-to-back testing). Where imperative programs were not available, the expected output was derived from first principles. Progress was monitored very carefully to ensure that the collection of the metrics could be performed in a controlled and accurate manner: the project began with a period of analysis before the language-specific parts of the project (design, implementation, testing and maintenance). All documentation and code was reviewed on a fortnightly basis. Language-specific test scripts were developed for all of the programs during the design stages.

The method used for this experiment was influenced by previous work [8] and by the guidelines provided by the DESMET Project [9].

4 Related work

The advocates of object-oriented programming languages claim that such languages have two main advantages [10–12].

- The facilities for data abstraction encourage the production of systems which can be easily modified.
- The facilities for polymorphism, dynamic binding and inheritance facilitate the production of libraries of classes which can easily be reused.

One of the aims of this project was to investigate the claims that robust, reusable and modular software can easily be produced in these languages.

The proponents of functional languages claim that they have the potential to provide solutions for many of the problems that confront the software engineer [13, 14]. This potential stems from the mathematical roots of such languages, together with facilities which are normally found in modern functional languages, such as polymorphism and higher order functions. In turn, these lead (it is claimed) to increased programmer productivity, amenability to formal verification and amenability to program transformation. Relatively little work has been published in the area of large-scale software development with functional languages [15–19].

Other researchers have investigated the use of functional languages for image processing [6]. However, their approach is very different to that described here, as it did not address the questions of measurement nor comparison with other languages. Comparisons between SML and C++ have been made using applications taken from numerical analysis [20], but this work is based on subjective criteria only.

Comparisons of Sisal and Fortran have been published [21–23] that discuss the performance, memory utilisation and compilation times of two equivalent sets of programs; they conclude that Sisal programs can run as fast as programs written in conventional languages. They state that programs written in functional languages are more concise, easier to write and to maintain.

A comparative experiment has been reported by researchers at Yale University [24], following an experiment which used a collection of languages (including Haskell, Ada and C++) to prototype a geometric region server. The Haskell implementation was found to be very concise (85 lines of code, compared with just over 1100 lines of C++), which the authors explain through the use of higher order functions, list processing functions and Haskell's simple syntax. The authors also report a large amount of code reuse during the evolution of the functional program and stress the utility of functional languages for prototyping, findings which our work supports.

5 Data collection

The following development metrics (DMs) were collected for every program:

□ the number of known errors found during testing (KEs).
□ the time to fix the known errors (TKEs), measured in minutes.
□ the number of modifications requested (MRs) during code reviews, testing and maintenance; this represents the number of changes that were requested excluding changes for fault clearance.
□ the time to implement modifications (TMRs), measured in minutes.
□ a subjective assessment of complexity (SC), provided by the system developer; this is based on an ordinal integer scale from 1 to 5, where 5 represents the most complex code.

Table 1 SML and C++ results: development metrics

development metrics	SML	C++	% difference
KE	66	37	78%
(KE/ncsl) × 1000	41	16	156%
MR	164	122	34%
SC	34	35	−3%
TKE	540	768	−30%
TMR	687	473	45%
DT	4324 min	3629 min	19%
TT	2349 min	1148 min	105%

These attributes were measured quantitatively; timings were measured in minutes, and a note was kept of each known error and modification request which occurred. The subjective complexity was given by the developer using an integer between 1 and 5. The programming languages were used during the design, implementation and testing of the programs. The development metrics were collected during implementation, testing and maintenance of the code, and were used in correlation analyses to determine the utility of the code metrics listed later in this section.

In addition to the above, the following development metrics were also collected: the time taken to develop the programs, measured in minutes (DT); and the time taken to test the programs, measured in minutes (TT). These times refer to the connect time, i.e. the actual elasped time.

The code metrics collected are listed below.

- ncsl: The number of non-comment non-blank source lines (non-comment source lines) for all codes written during the project. This was collected automatically after delivery of the source code.
- The number N^* of distinct functions that are called by the program. Each function is only counted once, no matter how many times it is called.
- The number L of distinct library functions that are called by the program. In this context, a library function is a standard function provided in an auxiliary module as part of the language or a general-purpose utility function (excluding those that are specific to the image analysis domain). For SML, the total for library functions includes the list processing functions provided as a standard part of the language, other standard functions (such as those dealing with arrays) etc. For C++ the total includes functions provided with the environment as standard library functions (such as those provided for input and output, standard maths functions etc.).
- The number D of distinct domain-specific functions where $D = N^* - L$, called by the program. A domain-specific function is defined as a function written specifically for the image analysis domain, defined within a module and also general utility functions.
- The depth of the function hierarchy chart.
- The number dec of function declarations that are specified within a program. This represents the size of a program's public interface.
- The number def of function definitions that are coded within a program. This represents the number of functions which have been implemented specifically for one program.

Table 2 SML and C++ results: code metrics

suggested indicators	SML	C++	% difference
ncsl	1595	2223	−28%
N^*	339	214	58%
L	100	26	285%
D	239	188	27%
L/N^*	0.29	0.12	142%
depth	51	35	46%
dec	52	57	−9%
def	176	109	61%

The code metrics were collated after testing, acceptance and maintenance of the programs. The code was tested rigorously through the use of test scripts with assertions. Only when all known errors were fixed was the code accepted as finished. The suggested indicators were produced by static analysis of the source code following final acceptance testing and a period of maintenance.

Twelve sets of algorithms were developed, together with a set of general-purpose library functions. During the SML development, 176 functions were defined, consisting of a total of 1595 non-comment source lines; this required 72 hours of connect time (staff hours) for the implementation and 39 hours for testing. The C++ development produced 109 functions, consisting of 2223 non-comment source lines. Implementation took just over 60 hours of connect time and testing required a further 19 hours. Full details of the results are shown in Tables 1 and 2.

6 Analysis and results

The aim of this analysis is to determine whether there is a significant difference in the quality of code produced using functional and object-oriented languages.

Earlier investigations [1, 2] used box plots to consider the distributions of the data sets and found them to be skewed. This was to be expected, as data collected from software development are rarely distributed normally [3, 8].

The Kruskal–Wallis one-way analysis of variance by ranks can be used to analyse data which are not normally distributed [25, 26]. The values from the data sets are ranked and the averages for each data set are calculated. The test assesses whether the difference between the averages of the ranks is significant. The results of the analysis of the corresponding SML and C++ programs are shown in Tables 3 and 4.

The statistically significant differences are listed below, together with the percentage differences which the SML programs exhibited for the overall totals:

- ☐ the time taken to test the programs, 105% more.
- ☐ the number of known errors per 1000 non-comment source lines, 156% more.
- ☐ the number L of library functions called, 285% more L/N^*, a measure of reuse, 142% more.
- ☐ the number of function declarations, 9% fewer.

Table 3 Kruskal–Wallis test results for the development metrics

development metrics	Kruskal–Wallis	probability
KE	2.20	0.14
(KE/ncsl) × 1000	4.20	0.04†
MR	0.17	0.68
SC	0.04	0.83
TKE	0.12	0.73
TMR	0.00	1.00
DT	0.65	0.42
TT	2.90	0.09§

† significant at the 5% level
§ significant at the 10% level

Table 4 Kruskal–Wallis test results for the code metrics

suggested indicators	Kruskal–Wallis	probability
ncsl	0.96	0.33
N^*	2.53	0.11
L	9.52	0.00†
D	0.30	0.58
L/N^*	4.86	0.03†
depth	0.07	0.79
dec	5.57	0.02†
def	2.46	0.12

† significant at the 5% level

All these differences were found to be significant at the 5% level, except for the difference in testing time, which was significant only at the 10% level. We consider each of these results in more detail below.

6.1 Development metrics

The SML code took a total of 39 hours to test, compared with only 19 hours for the C++ code. This can partly be explained by the fact that the C++ compiler produces executable object code which can be run from the command line, whereas the complete SML system must be reloaded to run the tests written in SML; although executable object code can be produced by the SML compiler used for this project, the procedure for doing so is not straightforward, and so this facility was not utilised. Another reason for the longer SML testing time is the number of tests which were run: 253 test cases were run for the SML code, compared with just 158 for the C++ programs. This may be due to the larger number of functions called by the SML programs (testing was organised on a per function basis). The longer testing time is also due in part to the larger number of known errors that were found in the SML software.

The number of known errors per thousand non-comment source lines was higher for the SML code than for the C++ code (41 compared with 16), a difference which is significant at the 5% level, and which necessitated the longer testing time. Values of 20 to 40 errors per one thousand lines of code [27] are typical of software development projects which use structured programming techniques and formal inspections. 50 to 60 errors per thousand lines are cited for software developed with unstructured designs and informal testing.

An analysis of the known errors reported is currently underway, and has revealed that 59% of the SML errors were associated with function calls (errors due to parameter values and types, missing parameters, use of the wrong function, incorrect function return types etc.). This is not surprising, considering that 61% more functions were defined within the SML programs. The higher error rate may be partly due to the fact that SML is a powerful high-level language and it is possible to achieve more functionality per line of code in such a language than in more conventional languages [14, 23, 24].

There were no statistically significant differences in the following development metrics: the number of known errors, the number of modification requests, the times taken to fix faults and make changes, development times and subjective complexities. Interestingly, the time to fix the known errors was found to be 30% less for the SML programs than for the C++ ones, even though the error rate was higher for the SML code and a C++ debugger was used. This suggests that the SML errors may have been less severe than the C++ ones; the developer readily agreed with this suggestion, and drew attention to the relative ease of debugging SML programs (see Section 8), lending weight to a thesis put forward by Hughes [14].

An investigation into the maintainability of the two sets of software is planned. The lack of any significant difference in the development time is also interesting, because it is clear that the edit–compile–execute loop takes much longer in SML than it does in C++, suggesting that programming in SML may increase programmer productivity, if only compiler efficiency could be improved.

6.2 Code metrics

Turning to the code metrics, there were no statistically significant differences in five of the seven suggested indicators (ncsl, N^*, D, depth and def). Of these, ncsl had previously been found to be closely correlated to the development metrics for both paradigms, confirming the lack of difference in the quality-type attributes measured by the development metrics. Although not statistically significant, some differences were apparent: the SML programs needed 28% fewer ncsl and defined 61% more functions. The shorter program length confirms previous findings [24]; the larger number of functions is not surprising because SML is a functional language and encourages adoption of the functional paradigm.

Both N^* and D had previously been found to be correlated to the number of modification requests for both paradigms, Neither of these metrics were found to be significantly different at the 5% level, reaffirming the lack of difference for the modification requests.

There is a significant difference in the number of library functions called; this showed little sign of being correlated with the quality-type development metrics in our previous work.

The metric L/N^*, the ratio of the number of distinct library functions called to the total number of distinct functions called, showed a statistically significant difference. The metric, which should lie between 0 and 1, will be

Table 5 Execution times of largest test cases, in seconds

program	number of pixels	*A* SML, s	*B* C++, s	*A/B*
PGM input	43 120	68.16	2.50	27.3
PGM output	43 120	36.90	3.57	10.3
PPM input	43 120	277.79	7.46	37.2
PPM output	43 120	115.15	5.37	21.4
convolution	43 120	126.63	40.74	3.1
Sobel	43 120	227.53	46.66	4.9
image to histogram	43 120	176.66	4.73	37.3
contrast stretch	43 120	9.57	4.55	2.1
D'Esopo	2 240	572.94	6.00	95.5
thinning	73 706	4 255.84	313.98	13.6
Gen. Hough transform	1 885 051	682.29	257.73	2.6

equal to 0 only if no reuse has taken place, but will tend towards 1 as the amount of reuse increases. In this study the SML code exhibited greater potential for reuse than the C++ code. However, it should be noted that these figures are affected by the large number of list processing functions (such as map, fold, hd, tl, take, drop, length etc.) routinely used by SML programmers. Such functions are commonly used in the composition of functional programs, in contrast with the C++ libraries which are usually used to provide more specialised functionality, such as the iostream, string and maths libraries; consequently this result is not surprising. Advocates of object-oriented languages claim reuse is one of the strengths of the paradigm; our research shows that it is an even greater strength for functional languages.

Although a significant difference was found for the number of function declarations, on inspection it was found that many of the programs developed (both SML and C++) had very few declarations (either zero or one). The significant difference was produced by a handful of C++ programs which had about 12 declarations each.

In conclusion, this investigation revealed only a few differences in the quality-type attributes of code written in functional and object-oriented languages. Further research is planned to determine whether these results would be repeated in a large-scale experiment.

7 Efficiency

A detailed investigation into comparative efficiency was not one of the aims of this project. However, some benchmarks have been recorded and are presented in this Section.

No special attempts were made to optimise any of the programs developed, as this may affect the quality of the code, and the extent of optimisation would be difficult to monitor. In addition, optimising code written in SML may have led to greater use of imperative language features, such as arrays, and this would have affected the conclusions of the investigation. Some imperative features (such as commands for input and output, arrays etc.) are used in a disciplined manner where circumstances dictate, as ignoring them completely would lead to a very contorted and unnatural coding style.

The programming languages SML and C++ were chosen as representative functional and object-oriented languages, partly because of the high-quality compilers that are available and distributed with liberal licensing agreements.* Compiler optimisation flags were not used during profiling because the effects of such flags are not always predictable or easily explicable, and optimisation was not fully implemented in the SML compiler used for the project.

7.1 Execution times

Table 5 shows the execution times for the largest test cases. The SML times are all greater than the C++ ones, by factors shown in the last column.

The majority of the differences in execution times can be explained by the choice of data structures used to hold the image data. An image was packaged as an ADT in SML and as a class in C++. The SML implementation stored the image as a list of lists, where each list represents a row of pixel values. All SML programs that used

* Standard ML, New Jersey, Version 0.93 is available from AT&T Bell Laboratories. G++ (gcc version 2.5.8) is supplied by the Free Software Foundation, MIT.

Table 6 Compilation and garbage collection times, in seconds

program	*A* SML, s, profiled	*B* GC SML(s)	*C* C++, s, unprofiled	*D* C++, s profiled	*A/D*
integer image object	31.23	25.90	3.22	3.24	9.6
real image object	31.67	26.14	3.22	3.24	9.8
PGM input	61.18	48.58	6.14	6.22	9.8
PGM output	62.70	47.74	3.30	3.26	19.2
PPM input	65.58	47.33	5.98	6.56	10.0
PPM output	72.05	56.94	3.22	3.22	22.4
convolution	61.56	46.63	3.22	3.20	19.2
Sobel	62.74	43.88	6.04	5.78	10.9
image to histogram	41.47	35.55	3.70	3.44	12.1
contrast stretch	62.57	42.06	3.66	3.64	17.2
D'Esopo	95.42	57.16	10.26	10.00	9.5
thinning	86.81	58.09	4.10	4.06	21.4
Gen. Hough transform	108.51	77.02	8.54	9.02	12.0

the image ADT utilised extensive list manipulation to process the given image one list (row of pixels) at a time.

The C++ class included member functions which provide facilities for the direct storage and retrieval of individual pixels (referenced by x and y co-ordinates plus the plane number within the image). The underlying implementation used dynamic memory allocation and pointer manipulation to store and locate individual pixels.

7.2 Compilation times

The results for the compilation times for the main SML and C++ programs are shown in Table 6. Note that each value is the average of five compilations, and that the times shown for the SML programs do not include the time taken for garbage collection, which is shown separately. The SML profiled compilation times are all greater than the C++ ones; the final column of the table shows the ratios of the two times.

8 Discussion

At the end of the project the developer was asked to record a subjective assessment of his experiences of programming with the two languages. The results are shown below.

8.1 SML: advantages

Programs are split into a number of small functions, which facilitates design through the use of function inputs and outputs. Standard list processing functions were found to be very helpful. SML code was found to be easier to debug than C++ code, mainly due to the language's strong typing. Automatic garbage collection means that programmers do not need to handle memory management.

8.2 SML: disadvantages

The SML code produced during this project had to be run from within the SML environment. SML lacks the flexibility of C++ when it comes to handling strings and files, and also when modelling the data ctypes required for some image processing routines. SML had a slower development turnaround, i.e. the programs generally took longer to compile and run than the equivalent C++ code.

8.3 C++: advantages

C++ is more efficient in terms of memory usage. Programmers are required to handle memory management using the constructors, destructors, etc. For image processing, it seemed easier to design the necessary classes than in SML. The C++ compiler produced stand-alone executable files which facilitated development and testing. There is better provision for screen, string and I/O handling in C++; the language is more powerful and flexible in this respect.

8.4 C++: disadvantages

The fact that the programmer has control of memory allocation means that this becomes a concern not found in SML: programs can crash due to segmentation faults etc. Although C++ can be used to write easy-to-follow object-oriented code, it is also easy to write large, convoluted functions, resulting in code which can sometimes be quite hard to understand and debug.

In conclusion, SML has proved to be very useful as a language for designing and prototyping image processing algorithms. However, pragmatism would lead to the choice of C++ over SML as an implementation vehicle for image processing.

9 Conclusions

In this investigation we have compared both development metrics and code metrics for sets of programs developed using languages from two different paradigms. Using algorithms taken from the image analysis domain, we performed the entire life-cycle for a suite of 12 programs, using both a functional and an object-oriented language.

We found no statistically significant differences in the development metrics, except for the length of testing time (where it was discovered that the code written using SML took twice as long to test as that written using C++) and the number of known errors per thousand non-comment source lines. Turning to the code metrics, we again found very few significant differences; no significant difference was found in the number of non-comment source lines (which had previously been found to be correlated with quality-type attributes for both paradigms). However, nearly four times as many library functions were called in the SML code than in the C++ code. This statistic is reflected in a metric for reuse which showed that the SML code exhibited one and a half times as much reuse as the C++ code. These results suggest that the use of functional programming languages may encourage a greater degree of reuse than object-oriented languages. It must be stressed that this is due (at least in part) to the large number of list processing functions which were utilised in the SML programs.

More significant, however, are the differences which were not found,; no significant differences were found in the following process metrics: the numbers of known errors, modification requests, the times to attend to these, a subjective measure of complexity, and the total development time. Our findings suggest that subjective preference may be a deciding factor when developers are forced to choose between programming languages.

10 Acknowledgements

The authors would like to acknowledge the support of the University of Southampton Research Committee for a grant to undertake this work.

11 References

[1] HARRISON, R., SAMARAWEERA, L.G., DOBIE, M.R., and LEWIS, P.H.: 'Estimating the quality of functional programs: an empirical investigation', *Inf. Softw. Technol.*, 1995, **37**, (12), pp. 701–707

[2] HARRISON, R., SAMARAWEERA, L.G., DOBIE, M.R., and LEWIS, P.H.: 'An evaluation of code metrics for object-oriented programs', *Inf. Softw. Technol.*, 1996, (3)

[3] FENTON, N.E.: 'Software metrics, a rigorous approach' (Chapman & Hall, London, UK, 1991)

[4] FENTON, N.E., and MELTON, A.: 'Deriving structurally based software measures'. *J. Syst. Softw.*, 1990, **12**, pp. 177–187

[5] DOBIE, M.R., and LEWIS, P.H.: 'Data structures for image processing in C', *Pattern Recognit. Lett.*, 1991, **12**, pp. 457–466

[6] KOZATO, Y., and OTTO, G.P.: 'Geometric transformations in a lazy functional language'. Int. Conf. on Pattern Recognition, The Hague, Netherlands, 1992

[7] HARTEL, P., and VREE, W.G.: 'Arrays in a lazy functional language — a case study: the fast Fourier transform'. ATABLE-92 Workshop, University of Montreal, 1992

[8] KITCHENHAM, B.A., PICKARD, L.M., and LINKMAN, S.J.: 'An evaluation of some design metrics,' *Softw. Eng. J.*, 1990, **5**, (1), pp. 50–58

[9] LAW, D.: 'DESMET methodology: overall specification'. DESMET Project deliverable 2.1, National Computing Centre, November 1992

[10] COX, B.J.: 'Object-oriented programming — an evolutinary approach' (Addison Wesley, Reading, Massachusetts, 1986)

[11] BOOCH, G.: 'Object-oriented design with applications' (Benjamin Cummings, Redwood City, California, 1991)

[12] MEYER, B:'Object-oriented software construction' (Prentice-Hall International, London, UK, 1988)

[13] TURNER, D.A. (1982): 'Recursion equations as a programming language' *in* DARLINGTON, HENDERSON, and TURNER (Eds.): 'Functional programming and its applications: an advanced course' (Cambridge University Press) pp. 1–28

[14] HUGHES, J.: 'Why functional programming matters,' *Comput. J.*, 1989, **32,** (2), pp. 98–107

[15] BUNEMAN, P., NIKHIL, R.S., and FRANKEL, R.: 'An implementation technique for database query languages,' *ACM Trans. Database Syst.*, 1982, **7,** (2), pp. 164–187

[16] AUGUSTSSON, L.: 'The Chalmers Lazy-ML compiler', *Comput. J.*, 1989, **32,** (2), pp. 127–141

[17] JOOSTEN, S., and RUPPERT, W.: 'Public transport in Frankfurt — an experiment in functional programming'. STAPLE Project (ESPRIT 891), Six monthly review, September 1989

[18] HARPER, R., and LEE, P.: 'Advanced languages for systems software'. Technical report, CMU-CS-94-104, 1994

[19] PAGE, R.L., and MOE, B.D.: 'Experience with a large scientific application in a functional language. ACM Functional Programming and Computer Architecture Conf., *FPCA 93*, Copenhagen, Denmark, June 1993, pp. 3–11

[20] SULLIVAN, S.J., and ZORN, B.J.: 'Numerical analysis using nonprocedural paradigms'. Technical report, University of Colorado at Boulder, 1993

[21] CANN, D., and FEO, J.: 'Sisal versus Fortran: a comparison using the Livermore loops'. Proc. Supercomputing 1990, IEEE, 1990, pp. 626–636

[22] CANN, D.: 'Retire Fortran? A debate rekindled'. Proc. Supercomputing 1991, IEEE, pp. 264–272

[23] CANN, D.C.: 'Retire Fortran?' *Commun. ACM*, 1992, **35,** (8), pp. 81–89

[24] HUDAK, P., and JONES M.P.: 'Haskell vs Ada vs C + + vs Awk vs · · ·' Technical report, Yale University, Department of Computer Science, 1994

[25] SIEGAL, S.: 'Nonparametric statistics for the behavioural sciences' (McGraw Hill, New York, 1956)

[26] SIEGAL, S., and CASTELLAN, N.J.: 'Nonparametric statistics for the behavioural sciences' (McGraw Hill, New York, 1988)

[27] DYER, M.: 'The cleanroom approach to quality software development' (Wiley, New York, 1992)

The paper was first received on 1 May 1995 and in revised form on 8 January 1996.

The authors are with the Department of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK.