Department of Computing Science and Mathematics
University of Stirling

Are Ours Really Smaller Than Theirs?

Simon P Booth
Simon B Jones

Technical Report CSM-141

November 1996

Department of Computing Science and Mathematics
University of Stirling

Are Ours Really Smaller Than Theirs?

Simon P Booth
Simon B Jones

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email s.p.booth@stir.ac.uk, sbj@cs.stir.ac.uk

Technical Report CSM-141

November 1996

Abstract

The claim is often made that functional programs are "more" expressive than their imperative counterparts. This paper examines this claim using a particular measure of (i) program length and (ii) programming language "level" (a measure of expressive power) both from the work of Halstead on software metrics.

This paper was presented at the 1996 Glasgow Functional Programming Workshop, held in Ullapool, July 1996. It has been accepted for publication in the proceedings (forthcoming).

1 Introduction

Halstead derived a number of measures of programs and other predictive statistics [10]. The two measures we will examine most closely in this paper are the *program length* and *language level* (both of these measures will be defined in section 2).

Although Halstead's measures use nice concrete features of programs (number of operators, number of operands, etc) they do not represent precise science and are the subject of some controversy [4] but these measures provide a more robust test of the "expressive" claim than the often quoted measure of number of lines [5], although Turner does give sound reasons to justify why functional programs are shorter. Hudak and Jones [11] also found that a prototype for a *Geometric Region Server* written in Haskell was considerably more succinct than equivalents written in traditional languages.

Most of Halstead's claims can only be verified experimentally. What gives us the confidence to proceed is that many experiments have been performed [3], [2], [7], [9], [6] and these confirm the claims made by Halstead. Halstead [10] summarizes the mean *language level* for a number of well known languages:

Language	*Mean(λ)*
English Prose	2.16
PL/1	1.53
ALGOL/68	1.21
FORTRAN	1.14
Assembler	0.88

This table above (apart from the English Prose entry) is drawn from the work of Zweben [12] which examines the same set of algorithms implemented by the same individual in each quoted language.

Halstead's claim is that the higher the mean language level the more *powerful* the language is. We prefer instead to say *'more expressive'* by which we mean that the same algorithm can be expressed more *economically.*

We intend in this paper to examine these values for the functional language Haskell [8]. Before proceeding to the results we give a little background to the work of Halstead and details of how we counted operands and operators.

2 Background

Before defining the program length and level we should note the following definitions that apply to a given program (or fragment) being analysed:

$$\eta_1 = \textit{the number of distinct operators that appear in the program} \quad (1)$$
$$\eta_2 = \textit{the number of distinct operands that appear in the program} \quad (2)$$
$$N_1 = \textit{the total number of operators} \quad (3)$$
$$N_2 = \textit{the total number of operands} \quad (4)$$
$$N = N_1 + N_2 \quad (5)$$

Operators are defined as the arithmetic operators, *begin..end, case..of, if..then, etc*; that is as syntactic program constructors. Operands are the variables the operators act on. See section 3 for the exact definitions we used.

The two values that we will examine in this paper have the following definitions:

A *prediction* of the total program length from the number of distinct operators and operands

$$\hat{N} = \eta_1 log\eta_1 + \eta_2 log\eta_2 \quad (6)$$

and the Language Level (λ) of the language as used in the program:

$$\lambda \;=\; \hat{L}^2 V \tag{7}$$

where the program *level*

$$\hat{L} \;=\; \frac{2}{\eta_1}\,\frac{\eta_2}{N_2} \tag{8}$$

and the program *volume*

$$V \;=\; N log(\eta_1 \;+\; \eta_2) \tag{9}$$

The *program length* may not seem a particularly important measure: it is easy to count the tokens, and to predict the total token count from the number of operators and operands may not seem particularly useful but Halstead makes a startling claim:

$$N \;=\; \hat{N} \tag{10}$$

and therefore that any programming language will exhibit the following relationship between the number of *distinct* operators and operands, and the number of actual operators and operands.

$$N_1 \;+\; N_2 \;=\; \eta_1 log \eta_1 \;+\; \eta_2 log \eta_2 \tag{11}$$

This equality has been verified by many experiments for the imperative languages Fortran, PL1 [10] and Pascal [6].

We are interested in this measure because we suspect that the "clean" syntax of Haskell, and the power of its operators, will provide shorter programs (measured by number of tokens) than imperative ones (all the previous work in the area that we are aware of is for imperative programs).

3 How We Counted

Haskell is a language that supports separate compilation and therefore a program would normally be broken down into a number of modules. Each module will consist of collections of function definitions or algorithms. To make our work as comparable as possible we use each individual function definition to represent an *algorithm* and do the counting at that level. We have also made the following assumptions:

- Type and data declarations are irrelevant, so are removed before counting
- Function type signatures are similarly removed before counting
- *if*. . .*then*, *let*. . .*in* and *case*. . . *of* are counted as single operators
- All function names are treated as operators. This is to be consistent with previous work. We believe that functions are *operands* for an implicit apply operator but previous work (with imperative languages) has dealt with subprogram names in calls as operators.
- A notional end-of-statement operator (`;`) is included for each statement although often in Haskell this is implicit (again, this is for consistency).

4 Preliminary Results

The results presented in this section are derived from the Happy [1] parser generator included as part of the `nofib` suite of programs [14] . Each function is treated as a separate "program" and N_1, N_2, η_1 and η_2 are calculated for each function. Only the modules `genutils.lhs`, `lalr.lhs`, `main.lhs`, and `producecode.lhs` have been analysed so far.

4.1 Program Length

Reexpressing equation (6) to incorporate a constant of proportionality α:

$$N_1 \ + \ N_2 \ = \ \alpha \ (\eta_1 log \eta_1 \ + \ \eta_2 log \eta_2) \tag{12}$$

previous experimental work has verified Halstead's prediction, that $\alpha = 1$, for most imperative languages. However, in our experiments with Haskell, the values we have observed for α are in the range 0.5-0.75. This indicates that the number of tokens is usually rather less than the number expected given the values of η_1 and η_2. So, on average, a Haskell program is rather shorter (measured by tokens) than an "equivalent" (i.e. same number of distinct operators and operands) imperative program. In fact, the number of tokens we actually need to write to construct a Haskell program may be as low as 50% of the imperative equivalent. One benefit is that there are less opportunities to make errors!

Why should this be? There may be both syntactic and semantic reasons.

Syntactically, one obvious difference occurs with function application. As noted earlier, the work with Pascal would count the following code fragment

```
f (x);
```

as the three operators `f`, `()`, `;` and the one operand `x`. Note that we can consider the `()` as representing *apply*. In Haskell we would almost certainly write this as

```
f x
```

which, under our counting scheme, is two operators `f`, `;` and one operand `x`. We *do not* count the *apply* as there is no symbol representing it. Artificially including an apply operator increases α but does not make it equal to 1. Indeed if we also choose not to include the implicit end-of-statement operator the value of α is even smaller.

On the semantic front, rephrasing (12), α will be small if the number of actual uses of each operator/operand is small relative to the number of distinct operators/operands used. In particular, this will be the case if the number of occurrences of each parameter and 'where-defined' identifier is low. We hypothesize that this *is* the case, as the semantic operators used in Haskell embody more computation than is expected in traditional imperative languages (i.e. they give more 'bang' per occurrence); however, we have not been able to quantify this yet. A further possibility is that the effect is due to the practice of introducing new identifiers quite freely, rather than 're-using' old ones (since assignment is not possible).

4.2 Language Level

The table below summarises the results from the modules examined so far, with the language level λ being calculated for each function definition:

Mean (λ)	2.63
Stan. Dev	5.13
Max (λ)	43.58
Sample Size	88

The maximum λ (43.58) is something of a outlier, the next largest λ is 19.22, but what is worth noting is the type of function that generates this very large value. We should also note that this value is also making the standard deviation quite large. The function that generates the λ of 43.58 is the function *argFns* in the main module. It is reproduced below:

```
argFns "-infile"      = flag DumpInFile
argFns "-lex"         = flag DumpLex
argFns "-parse"       = flag DumpParse
argFns "-mangle"      = flag DumpMangle
argFns "-lr0"         = flag DumpLR0
argFns "-action"      = flag DumpAction
argFns "-goto"        = flag DumpGoto
argFns "i"            = flagWithOptArg OptInfoFile
argFns "-info"        = flagWithOptArg OptInfoFile
argFns "-template"    = flagWithArg OptTemplate
argFns "-magic-name"  = flagWithArg OptMagicName
argFns "v"            = flag DumpVerbose
argFns "-verbose"     = flag DumpVerbose
argFns "-lookaheads"  = flag DumpLA
argFns "g"            = flag OptGhcTarget
argFns "-ghc"         = flag OptGhcTarget
argFns "a"            = flag OptArrayTarget
argFns "-array"       = flag OptArrayTarget
argFns "o"            = flagWithArg OptOutputFile
argFns "-outfile"     = flagWithArg OptOutputFile
argFns _              = const Nothing
```

This is essentially a *case* statement that handles the command line options when Happy is started. (The function that generates the λ of 19.22 is of a similar type). *argFns* has 7 operators: `argFns, flag, flagWithOptArg, flagWithArg, const, =, ;` that are used on 84 occasions and 37 operands which occur 42 times. This gives values for V and $\hat{L}$ of 687.89 and 0.25. So we have a relatively large program (as measured by volume) with a high $\hat{L}$. $\hat{L}$ is high because of the ratio of η_2 and N_2: each operand occurs at most twice and most only occur once giving a very efficient use of operands and consequent high language level. It should be noted that when measures of program complexity are examined ([13]—see section 5) *case* statements cause the measures to *overestimate* the complexity; *case* structures are clearly a very particular programming construct.

Before we discuss these results in relation to imperative languages we should investigate what effect the size of the program (as measured by N) has on the language level. We wish to do this to eliminate any potential bias caused by *small* functions like (this is only done in an attempt to make comparsions with imperative programs more meaningful: we would not expect to find such *small* functions in the imperative domain):

```
startRule = singletonSet (0,0)
```

Exactly where to choose the cut off is somewhat arbitrary and we present the relevant values for a range of N. (In each column we exclude programs smaller than the specified N.)

Program Size	N>0	N>10	N>20	N>30	N>40
Mean (λ)	2.63	2.82	2.45	2.65	2.95
Stan. Dev	5.13	5.68	6.36	7.07	7.99
Max (λ)	43.58	43.58	43.58	43.58	43.58
Sample Size	88	71	46	37	29

As N increases it is quite clear that the outlying value for λ (*argFns*) is dominating the results. There are two ways to deal with this: increase the sample size or eliminate it from the results (classifying it as an outlier that is causing a bias in the results). The table below summarises the same results with the large λ removed:

Program Size	N>0	N>10	N>20	N>30	N>40
Mean (λ)	2.16	2.24	1.53	1.52	1.50
Stan. Dev	2.62	2.89	1.42	1.51	0.44
Max (λ)	19.22	19.22	7.83	7.83	7.83
Sample Size	87	70	45	36	28

We can see that the large λ was dominating the results from N>20 and upwards. To justify any claims that Haskell has a higher language level than the imperative it would appear that we "need" to include only the results for functions with ten or more tokens, giving a language level of 2.24. It is equally clear that as N increases that the language's level falls! This is due to the decision to treat functions as operators, as, in (7), the number of distinct operators appears as η_1 in the denominator and $log\eta_1$ in the numerator. Not surprising in a functional language we would expect there to be a preponderance of function calls! For comparative purposes we present the calculations for the above table but counting functions as *operands*

Program Size	N>0	N>10	N>20	N>30	N>40
Mean (λ)	7.27	7.34	7.25	8.20	9.52
Stan. Dev	17.68	20.31	24.95	27.80	31.37
Max (λ)	171.97	171.97	171.97	171.97	171.97
Sample Size	94	71	46	37	22

(The sample size has grown because some functions previously had no operands and therefore could not be analysed but changing the counting scheme means that they now have operands and so λ can be calculated.) *argFns* is still the outlying value and now generates a considerably larger λ value. The main point to notice is the large values for the means of λ (again as N increases *argFns* is dominating the results); on this occasion its removal changes the language level to 5.50 (with stan.dev 4.25). Whilst treating the functions as operators may not agree with previous work in the functional domain this seems an entirely natural way to deal with functions—they are, after all, operands to the implicit operator *apply*.

5 Further Work

At present we have only analysed work done by two programmers [1] and to draw more general conclusions it would seem wise to analyse rather more of the `nofib` suite of programs (particularly those programs in the Real Set). This will also have the benefit of enlarging the sample size. At present we cannot emphasize too strongly that the numbers produced in the last section are preliminary and will be amended when more Haskell programs have been analysed.

Comparisons between various functional languages would seem to be an interesting area of study as we can control the counting scheme and consistency is assured. Whether we can use the mean λ to compare between different types of languages is an open question—certainly one must be careful to compare like with like.

There are other well-known metrics for programs. Perhaps the most widely quoted is McCabe's metric [13]. This metric attempts to describe an imperative program in terms of its *cyclomatic complexity*. The underlying theory is drawn from graph-theory but the resulting value for the cyclomatic complexity appears to be very useful—essentially the value derived represents the number of different paths through the program (called the *basis path set*). The different paths through the program can be considered to provide a useful set of tests of the program. The question that we would like to investigate is whether this measure is lower for functional programs (some early work we have done would appear to suggest this is the case—implying that the program are less complex and therefore easier to test and maintain).

6 Conclusions

We started out in this investigation of the area, that is sometimes called "Software Science", to see how well it worked with functional languages. In particular, whether it provides any evidence that programs written in functional languages are shorter (in terms of token count) and more expressive (as measured by language level) than imperative languages. The motivation for the latter part of investigation is the table presented in the introduction—it appears to provide evidence that PL/1 is more expressive than Algol 68 which in turn is more expressive than Fortran, and that Fortran is more expresive than assembler—results that would confirm one's intuition about imperative languages. Although as we noted in the introduction the whole area of "software science" is the subject of some controversy.

The initial results appear to confirm that functional languages are shorter than their imperative equivalents. Whether they are "more expressive" remains an unanswered question.

We must recall that we are using a "science" developed with imperative languages, and that all the confirming results (as to the accuracy of the proposed measures) are drawn from that domain. In the functional domain the measures of "software science" appear to break down. In section 4.1 we noted that equation 6 is incorrect for the Haskell programs we have examined. Whilst this indicates that the operator and operand count should not be be used in Haskell to predict program length using Halstead's formula. It does, nevertheless, indicate that the token counts are lower than would be expected in the imperative domain.

The results for language level (the measure of expressiveness) are rather more difficult to interpret and appear to add weight to the non-applicability of "software science" in the functional domain (whatever one's opinion for the imperative). It might improve matters if we increased the sample size and included the work of a wider range of programmers but these initial results are not encouraging.

Also, as already stated, two different counting schemes appear have been used to compare imperative languages (whole programs and algorithms) and procedure calls are treated as operands which, in effect, get counted twice if the language uses the "standard" `f (x);` syntax whereas in the functional domain we only count `f x` once unless we deliberately count the implied apply. More importantly there is the question of how we treat functions themselves: as operators or operands. Our view is that they are operands to the implicit apply.

Whilst "software science" might provide useful comparators between functional languages, it does not appear to provide any useful way of comparing functional and imperative languages (apart from the token count as this is such a striaght-forward measure and as a measure is superior measure than number of lines as it is unlikely to be influenced by different layout styles). Currently it is not our intention to extend this investigation but we hope to examine other forms of software metrics to see if any useful measures of program complexity and size are available in the functional domain.

Overall, though, it would appear that function programs are shorter (in terms of tokens) than imperative given the same number of operators and operands. Of course, it may well also be the case that functional programs also require less operators and operands to express the same algorithm and, if this is the case, the token count will be even lower—for an "equivalent" program. Overall, though, we must conclude that "software science" appears to offer little when used to provide metrics in the functional domain.

References

[1] Gill A and Marlow S. Happy—the parser generator for Haskell . http://www.dcs.gla.ac.uk/fp/software/happy/.

[2] Feuer AR and Fowlkes EG. Relating computer program maintainability to software measures. In *Proceeding of the 1979 National Computer Conference*, 1979.

[3] Feuer AR and Fowlkes EG. Some results from an empirical study of computer software. In *4th International Conference on Software Enginnering*, September 1979.

[4] Levitin AV. How to measure software size, and how not to. In *10th International Computer Science and Application Conference*, October 1986.

[5] Turner DA. Recursion equations as a programming language. In Henderson P Darlington J and Turner DA, editors, *Functional Programming and its Applications*, pages 1–28. Cambridge University Press, 1982.

[6] Johnston DB and Lister AM. An experiment in software science. In Tobias JM, editor, *Language Design and Programming Methodology*, number 79 in LNCS. Springer-Verlag, 1980.

[7] Fitos GP. Vocabulary effects in software science. In *4th International Conference on Computer Software and its Applications*, October 1980.

[8] Peyton Jones S Hudak P and Wadler P et al. *Report on the Programming Language Haskell*, March 1992.

[9] Lipow M. Comments on "estimating the number of faults in code" and two corrections to published data. *IEEE Trans. Software Engineering*, 12(4):584–585, 1986.

[10] Halstead MH. *Elements of Software Science.* Elsevier, 1977.

[11] Hudak P and Jones MP. Haskell vs. Ada vs. C++ vs. awk. ... an experiment in software prototyping productivity. Department of Computer Science, Yale University, July 1994.

[12] Zweben SH. *The Internal Structure of Algorithms.* PhD thesis, Purdue University, 1974.

[13] McCabe T. A software complexity measure. *IEEE Trans. Software Engineering*, 2(6):308–320, 1976.

[14] Partain W. The `nofib` benchmarks suite of Haskell programs . http://www.dcs.gla.ac.uk/fp/software/ghc/nofib.html.