Automatic analysis of functional program style
ASWEC'96
IEEE Copyright

Greg Michaelson
Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton, Scotland, EH14 4AS
greg@cee.hw.ac.uk

Abstract

Functional programming style is discussed and seven simple semantic style rules, based on program transformation, are enunciated. An automatic style analyser developed from these rules is then presented.

1 Introduction

Good programming style is deemed important to communicate programs effectively to other people, to optimise implementations and to minimise error proneness. However, programmming style is notoriously difficult to characterise and there are substantial differences of opinion about just what constitutes good programming style.

We adopt the viewpoint that style is not concerned with *what* is said but rather with *how well* it is said. In the context of programming, we assume that a program is correct in the sense of producing the appropriate outputs from inputs, whether demonstrated formally or by convincing testing. Thus, program style is concerned with how effectively an input/output process is expressed, regardless of what that process actually is.

There is a temptation to overload style and make it responsible for all aspects of programming. Our viewpoint sees style as subsequent to design decisions, concerned with presentation rather than content. For example, in a system to count ASCII characters in text files, the representation of character/count associations in linked lists rather than direct indexed arrays is a poor design decision rather than an instance of poor style. In contrast, writing an iterative quicksort is stylistically dubious. To make an analogy with natural language, style is concerned with the organisation of chapters, paragraphs and sentences, not the plot.

This analogy should not be taken too far: computers are far less forgiving than humans of imprecision and differences in usage. For example, while purists may fulminate against split infinitives or sentences starting with conjunctions, these are no hindrance to spoken communication and an annoyance, rather than a source of ambiguity or error, when written. Indeed, a distinctive style, where "rules of good style" are bent to create an aesthetic effect, is considered a hallmark of a creative writer. In contrast, in computers the compiler and run-time system are the ultimate arbiters of program acceptability. Anything that does not correspond precisely to the language definition, be it explicit or implicit in the implementation, will be rejected. Furthermore, programming languages are much simpler than natural languages, giving far less scope for stylistic variation. Nonetheless, style inconsistencies seem subjectively to have much more impact on the intelligibility of programs than of novels.

We see succinctness, economy of expression, as central to style. This should not be confused with brevity, the soul of obfuscation as well as of wit: it is often poor style to say too little as well as too much. Succinctness is related to the expressive power of languages: those with well developed orthogonal abstraction mechanisms tend to support succinct programmming. Despite continuing advances in optimising compilers, there is still an inverse relationship between expressive power in programming languages and efficiency of implementation. For example, consider recursion, structured programming and pattern matching. Each enables succinct programs yet each has more efficient but less succinct equivalents, namely iteration, gotos and structure selection. Thus, we necessarily view implementation aspects of programming style as at best secondary to the effective communication of programs between people.

2 A linguistic characterisation of style

Programming languages are defined in terms of a lexicon, syntax, semantics and pragmatics. It is a useful starting point to consider how style relates to each of these.

There are two approaches to style: relative and absolute. Relative style is concerned with assessing something against an appropriate exemplar whereas absolute style is based on generic, domain independent rules. Here we will consider absolute style.

The lexicon specifies the symbols - the smallest meaningful units in a language. In programming, a programmer only has control over the construction of identifier symbols for variables. The use of meaningful identifiers in programming is a fundamental rule of thumb but not amenable to automatic checking. There have been attempts to prescribe minimum lengths for identifiers but these are self defeating: in general, there is no connection between an identifier's length and its meaningfulness. Furthermore, "meaningless" small identifiers like `i`, `j` and `k` are used universally for loop variables, following mathematical practice.

The syntax specifies valid grammatical structure - well formed symbol sequences. In programming, a programmer must stick exactly to the syntax and only has control over how a program is laid out. There have been many attempts to specify layout rules for free format languages, for example to regularise indentation for blocks, but for any given language no one layout style has found general favour. For some languages, for example KRC and occam, layout is prescribed. Many layout checking and pretty printing systems for different languages have been developed. In practice, there is probably no need for a common layout style provided each programmer uses one style consistently.

The semantics specifies the meaning of well formed symbol sequences. Here, style is concerned with the use of appropriate constructs. For imperative languages, which typically offer a wide range of semantically equivalent constructs, this is a source of endless theological disputes. For example:

> Are gotos to be discouraged?
> Are FORs better than WHILEs for sequenced structure access?
> Is recursion preferable to iteration?
> How many parameters should a sub-program have?

Dijkstra[3] answers the first question eloquently in the affirmative: the basis of structured programming. It is much harder to address the other questions rationally without some overarching guiding principles. Typically, such questions also conflate issues of implementation efficiency with ease of human understanding. In principle, equivalent semantic constructs should be compilable to the same low level code. In practice, compilers and optimisers are not yet sufficiently adept. Arguably, if the least complex algorithm has been constructed then the mode of its expression is arbitrary.

Pragmatics is concerned with the context of use. In programming, there may be differences in the support environment, for example in the provision of libraries, but again a programmer has little control over these. In style terms, however, we may view the use of comments, assertions and type annotations as pragmatic issues. Comments provide a natural language description of program components to ease human understanding. Like meaningful identifier use, the use of comments is generally thought to be good style. However they are not amenable to automatic analysis other than with crude checks for their presence and size. Assertions specify logical requirements that should be met at different points in programs. They are most valuable where they are used to prove programs correct - not a feature of any extant language implementation - or where they are implemented as additional run-time checks. Otherwise they are effectively comments. Some polymorphic typed languages, for example HOPE, require explicit type annotations. In others, for example Haskell, they are optional. Requiring type annotations is useful for teaching, to ensure that students understand typing, but frustrating for general use. A type checking system can often deduce the types of variables from the context of their use: where it cannot the variables must be typed explicitly. Thus optional type annotations also serve as comments.

Our approach is semantics oriented and discussed further below.

3 Background

Programming language textbooks have curiously little to say about program style beyond the use of meaningful identifiers and layout. Even the classic books by Kernighan and Plauger[12] and Ledgard, Hueras and Nagin[14] are essentially collections of maxims. Both range across the whole gamut of linguistic style characteristics outlined above; neither enunciates a consistent approach to ensuring good style.

There have been numerous attempts to characterise programming style for program assessment purposes both in education and industry. Style checking based on various criteria has been incorporated into program analysis systems. Here we survey salient work.

Some authors seem to question the possibility of formalising style at all. For example, Miller and Peterson[17] describe an approach to program evaluation for use by instructors. They allocate 10% of marks to comments, 10% to meaningful data names, 10% to indentation and 5% to design modularity. Other marks are given for output clarity(35%), program length(5%), and the nebulous "output embellishments"(5%) and "exemplary program style and clarity"(10%). The final 10% is for the flowchart. Olson[18], in contrast to Miller and Peterson's "analytic" method, proposes a holistic approach to grading. Here, the instructor is to assess the program as a whole using a set of very loose guidelines. Olson found little correlation between holistic and analytic marks, with holistic marks being consistently higher. He comments that the approach relies on the assumption that "computer science teachers recognise good programs when they see them" and that it provides little diagnostic information. Howatt[10] criticised Olson for failing to justify his holistic guidelines. He attempts to merge analytic categories with holistic criteria for each. Style is one such category, receiving 15% of overall marks, with "Well formed, understandable code; appropriate use of language capabilities" as desirable holistic criteria. We think that the holistic approach is far too vague to be useful: our work is based on the premise that programming style is explicitly quantifiable.

Marca[15] presents style guidelines for Pascal. Most recommendations, beyond the use of meaningful identifiers, are syntactic, for example the use of parenthesise to clarify expression evaluation order and general layout rules. Alas, one example suggests that `if EndOfFile=false` should be changed to the preferable `if EndOfFile=true` with the `then` and `else` actions swapped on the grounds that a negation is harder to understand than a positive condition. However the comparison with a boolean value is a classic semantic style error.

Gillett[6] developed the Interactive Program Advising System to comment on FORTRAN programs. Most checks are for low level semantic "conceptual errors" for example failing to simplify arithmetic expressions or eliminate common subexpressions or using null IF statements. This system is interesting in suggesting how the program might be changed to avoid the error. S.S.Robinson and Soffa's system[22] for FORTRAN is based on a combination of Halstead's metrics to count various program features, for example number of unique variables, operators, data structures and sub-programs, and semantic checks for example to favour FORs over WHILEs and the use of one 2D rather than several 1D arrays. Their hope is that students will "learn how to write efficient programs that make correct use of control structures". S.K.Robinson and Torsun[21] also use Halstead's metrics in their FORTRAN checker.

Rees' influential STYLE system[20] evaluated Pascal programs on a mix of lexical(length of identifiers, varieties of identifier names), syntactic(line length, layout) and pragmatic(comments) criteria. It also checked for goto use. Rosenthal[23] developed an awk script to carry out Rees' checks which Meekings[16] further refined. Berry and Meekings[2] then changed the system to check C programs and ran it on 80,000 lines of code from well known UNIX utilities. Overall, the code received low style marks. The authors were amused to note that the lowest score went to a "C beautifier" program. In general, they found very low use of comments and high use of gotos. However, Harrison and Cook[9] then applied Berry and Meeking's system to 20 malfunctioning C modules containing 35,000 lines of code. Their analysis showed that the style measure had no relationship to the error frequency. They criticise Rees' criteria on the grounds of adequacy and completeness, and suggest that attention should be turned to how style rules affect human program comprehension. Finally, Hannemyr[7] criticises Rees for seeking a "symbolic solution to a semantical problem" and notes "our profession's blind faith in instrumentalism".

Redish and Smyth[19] propose six criteria for style analysis i.e. economy, modularity, simplicity, structure, documentation and layout. Like Rees' STYLE and the Halstead based systems described above, their AUTOMARK for FORTRAN 77 is based almost entirely on low level counts of program constructs. Lake and Cook's[13] STYLE system for Pascal is also based on Redish and Smyth's criteria.

Van Verth and Ralston[25] criticise systems based on feature counting and propose the use of Oviedo's complexity measures, based on control graph analysis. They have used their approach to implement a Pascal grader and claim that a low complexity measure is indicative of good style. Nonetheless, this is still feature counting, albeit at a higher level.

In summary, almost all work has focussed on non-semantic aspects of style and automatic style analysis is based primarily on construct counting. We agree with Hannemyr that semantic style considerations are central but think that if they can be made explicit then they can be subject both to symbolic analysis and measurement.

4 Dromey's software product quality model

Recently, Dromey[5] has elaborated a hierarchical property model for assessing software product quality, in response to the international standard ISO-9126 Software Product Evaluation Characteristics[11]. He identifies correctness and style properties as the fundamental contributors to quality and relates these to the structural forms of programs. This enables him to locate the source of poor quality at the appropriate lowest structural level in a program. He enunciates categories of properties termed correctness, concerned primarily with proof and termination, structural, concerned with low level constructs, modularity, concerned with high level constructs, and descriptive, concerned with specification and documentation. Thorough descriptions of each are provided with examples.

Dromey's framework is strongly influenced by the Floyd/Hoare/Dijkstra imperative program proof and synthesis tradition[4], leading to a pleasing focus on semantic issues in the correctness, structural and modularity categories. The descriptive properties are pragmatic(specifications, comments) and lexical(meaningful identifiers). However, where semantic properties are motivated by program proof and termination concerns, primarily the correctness properties, they are not amenable to automatic analysis but require substantial computer assistance for hand assurance of programs. Most other properties are suitable for automatic assessment, in particular many of the structural properties motivated by structured programming and the modular properties concerned with issues of cohesion and coupling. However, some non-formal properties make reference to the problem a program is intended to solve and are not automatable, for example those structural and modular properties to do with the necessity or abstractness of constructs.

Despite our reservations, Dromey's model represents a significant advance in characterising imperative program style.

5 Semantic style and functional programming

Functional languages have been proposed widely as a basis for rigorous software development. Pure functional languages lack time ordering through assignment, leading to relative ease of program proof and transformation. They are also seen as valuable pedagogically in their own right, and are taught on an increasing number of courses[8].

Functional languages are more succinct than many of their imperative brethren. In particular, case structuring with pattern matching enables close associations between the structures of recursive data structures and the functions that manipulate them. However, functional languages still provide alternative ways of expressing the same intention. For example, Haskell has case structured pattern matching and guarded patterns as well as an explicit conditional expression. For example, many functional languages allow structure access with explicit selectors as well as through pattern matching.

Our interest in functional programming style is strongly motivated by concern for common mistakes and misapprehensions in elementary functional programs written by novices. We think that semantic style errors in programming arise from conceptual misunderstandings. If style is amenable to formal characterisation then the learning of programming can be supported by a program analysis system which identifies style errors and suggests how they can be overcome.

We do not claim to provide an exhaustive taxonomy of functional programming style akin to Dromey's: we do not consider correctness or descriptive properties, and our focus on programming in the small reduces the relevance of modular properties. Furthermore, the absence of assignment renders many structural and modular properties irrelevant. We are left with what are effectively structural properties which are amenable to automatic analysis.

The principle of compositionality[24] was enunciated for language semantics: a construct should be defined in terms of the meanings of its sub-constructs. Compositionality is related to Jackson's correspondence principle which Dromey cites as the basis of his "resolved" structural property. This is a fruitful starting point for elucidating semantic style rules for functional languages which provide case structure with pattern matching. The overall structure is identified in the bound variable pattern on the left of a case. The sub-structures are then manipulated in the case body. For a recursive structure, this enables the clear differentiation of base and recursion cases corresponding to base and recursive components of the structure. This approach is certainly not universally applicable: many problems do not have strictly compositional solutions. Nonetheless, compositionality enables a first focus on the semantic component of functional program style.

In the following sections, we present style checking rules for two aspects of compositionality. We then discuss rules for introducing "partial application", for removing unused variables in patterns and for simplifying boolean comparisons and results. These latter rules are also semantic in focus and motivated by a desire for economy of expression.

Rules are presented here as schematic rewrite transformations enabling the correction as well as the identification of style errors. It would be relatively simple to prove formally that each rule preserves meaning: this is not done here.

5.1 Compositionality and lists

Consider the list type which usually provides an empty list and a concatenation of a head and a tail:

```
list -> []
list -> head::tail
```

In strongly typed languages, if the head is of some element type then the tail must be a list of that type or empty.

An arbitrary list may be processed with a base case for an empty list and a recursion case for a non-empty list:

```
fun process list =
      if list=[]
      then base case
      else recursion case using
             (hd list) and (tl list)
```

where `hd` selects the list head and `tl` selects the list tail.

An alternative to a conditional expression with an explicit comparison with the empty list is to separate out the base and recursion cases through pattern matching:

```
fun process [] = base case |
    process list =
      recursion case using
        (hd list) and (tl list)
```

Similarly, explicit list selection in the recursion case can be replaced with pattern matching:

```
fun process [] = base case |
    process (head::tail) =
      recursion case using head and tail
```

We think that this compositional approach is preferable: the cases and patterns make explicit the overall structure of the data to be processed and the function bodies show the processing of sub-structures.

5.2 Case structure from constant comparison

Our first rules generalise the list example, to introduce case structure from a conditional expression by replacing any explicit comparison between a variable and a constant in the condition with pattern matching:

```
fun name pattern with var =
 if var=const
 then expression 1 with var
 else expression 2 with var ==>

fun name pattern with const =
      expression 1 with const |
    name pattern with var = expression 2
```

For example:

```
fun sum l =
     if l=[]
     then 0
     else sum l = hd l+sum (tl l) ==>

fun sum [] = 0 |
    sum l = hd l+sum (tl l)
```

We identify a special case for a condition which is a boolean variable, as the variable need not be retained:

```
fun name pattern with var =
 if var
 then expression 1 with var
 else expression 2 with var ==>

fun name pattern with true =
      expression 1 with true |
    name pattern with false =
      expression 2 with false
```

For example:

```
fun not x =
 if x
 then false
 else true  ==>

fun not true = false |
    not false = true
```

5.3 Patterns from explicit structure selection

Our second rule replaces explicit list selection with pattern matching:

```
fun name pattern with var =
      expression with (hd var) and
                      (tl var) and var ==>

fun name pattern with (h_var::t_var) =
      expression with h_var and t_var
                     and h_var::t_var
```

For example:

```
fun sum [] = 0 |
    sum l = hd l+sum (tl l) ==>

fun sum [] = 0 |
    sum (h_l::t_l) = h_l+sum t_l
```

5.4 Partial application

Functional languages enable the definition of nested functions of single bound variables, known as curried functions. Such functions may be "partially applied" by specialising outer arguments. For example:

```
fun add x y = x+y
```

is a function of `x` which returns a function of `y` which adds `x` and `y`. Thus:

```
val inc = add 1
```

is a function of `y` with `x` bound to `1`.

Inexperienced functional programmers often provide an unnecessary explicit inner bound variable in such definitions:

```
fun inc y = add 1 y
```

This may be remedied by:

```
fun name patterns pattern =
      expression pattern ==>

fun name patterns = expression

fun name pattern =
      expression pattern ==>

val name = expression
```

provided no identifiers in *pattern* appear in *expression*.

5.5 Unused variables in patterns

Many functional languages provide a wildcard pattern which matches anything. This may replace a variable in a pattern which is not used subsequently, to simplify patterns and emphasise significant elements in structures:

```
fun name pattern with var =
      expression without var ==>

fun name pattern with _ = expression
```

For example:

```
fun first (x,y,z) = x ==>

fun first (x,_,_) = x
```

5.6 Boolean comparison and results

A common novice excess in many languages is to test explicitly for a boolean value using comparison. This can be replaced with the compared expression:

expression`=true ==>` *expression*

expression`=false ==> not` *expression*

For example:

```
fun ispos x = (x>0)=true ==>

fun ispos x = x>0
```

A related foible is to return a boolean value from a conditional expression where the condition itself will suffice:

`if` *condition*
`then true`
`else false ==>`

condition

`if` *condition*
`then false`
`else true ==>`

`not` *condition*

For example:

```
fun isneg x =
 if x<0
 then true
 else false ==>

fun isneg x = x<0
```

These errors seem to suggest a misunderstanding about the relationship between conditions and boolean values. Conditions are mainly used in the context of decisions rather than to return values, and non-boolean values must be constructed and tested explicitly.

6 Implementation

A style checker for a pure functional subset of Standard ML was written in C using the above rules. It parses SML functions to construct a tree and then checks each rule. Where a rule applies, the tree is transformed and pretty printed along with a terse message.

For example, to replace conditioned constant comparison with case structured pattern matching:

```
fun pred x =
 if x=0
 then 0
 else x-1

CHECK 6:
 CONVERT if <var>=<const> TO PATTERN
fun pred x =
 if x=0
 then 0
 else x-1 ==>

fun pred 0 = 0 |
    pred x = x-1
```

To replace a boolean variable as a condition with case structured pattern matching:

```
fun and x y =
 if x
 then y
 else false

CHECK 5: CONVERT if <var> TO PATTERN
fun and x y =
 if x
 then y
 else false ==>

fun and true y = y |
    and false y = false
```

To replace structure selection with patterns:

```
fun ordered [] = true |
    ordered [v] = true |
    ordered l =
     (hd l)<(hd (tl l)) andalso
      ordered (tl l)

CHECK 7: CONVERT hd/tl USE TO PATTERN
l ==> h_l::(h_t_l::t_t_l)

CHECK 7: CONVERT hd/tl USE TO PATTERN
hd l ==> h_l

CHECK 7: CONVERT hd/tl USE TO PATTERN
hd (tl l) ==> h_t_l

CHECK 7: CONVERT hd/tl USE TO PATTERN
tl l ==> h_t_l::t_t_l

fun ordered [] = true |
    ordered [v] = true |
    ordered (h_l::(h_t_l::t_t_l)) =
     h_l<h_t_l andalso
      ordered (h_t_l::t_t_l)
```

To introduce partial application:

```
fun sizes l = map size l

CHECK 3: INTRODUCE PARTIAL APPLICATION
fun sizes l = map size l ==>

val sizes = map size
```

To remove unused pattern variables:

```
fun seconds [] = [] |
    seconds [v] = [] |
    seconds (h1::(h2::t)) =
     h2::(seconds t)

CHECK 4:
 REMOVE UNUSED VARIABLE IN PATTERN
v ==> _

CHECK 4:
 REMOVE UNUSED VARIABLE IN PATTERN
h1 ==> _

fun seconds [] = [] |
    seconds [_] = [] |
    seconds (_::(h2::t)) =
     h2::(seconds t)
```

To remove comparison with a boolean constant:

```
fun alpha ch =
 (ch>="a" andalso ch<="z")=true

CHECK 2: REMOVE COMPARISON WITH BOOLEAN
(ch>="a" andalso ch<="z")=true ==>

ch>="a" andalso ch<="z"

fun alpha ch = ch>="a" andalso ch<="z"
```

To remove a condition returning boolean results:

```
fun notdigit ch =
 if ch>="0" andalso ch<="9"
 then false
 else true

CHECK 1:
 REMOVE if <cond> then true else false
if ch>="0" andalso ch<="9"
then false
else true ==>

not (ch>="0" andalso ch<="9")

fun notdigit ch =
     not (ch>="0" andalso ch<="9")
```

Rules are applied repeatedly until none succeed. Finally, consider:

```
fun count0 l =
 if l=[]
 then 0
 else
  if (hd l)=0
  then 1+(count0 (tl l))
  else count0 (tl l)
```

which counts how often `0` appears in a list of integers.

First of all, the checker identifies that the explicit comparison of the list argument against the empty list may be replaced with two cases:

```
CHECK 6:
 CONVERT if <var>=<const> TO PATTERN
fun count0 l =
 if l=[]
 then 0
 else
  if (hd l)=0
  then 1+(count0 (tl l))
  else count0 (tl l) ==>

fun count0 [] = 0 |
    count0 l =
 if (hd l)=0
 then 1+(count0 (tl l))
 else count0 (tl l)
```

Next, explicit use of `hd` and `tl` may be replaced by patterns:

```
CHECK 7: CONVERT hd/tl USE TO PATTERN
l ==> h_l::t_l

CHECK 7: CONVERT hd/tl USE TO PATTERN
hd l ==> h_l

CHECK 7: CONVERT hd/tl USE TO PATTERN
tl l ==> t_l

CHECK 7: CONVERT hd/tl USE TO PATTERN
tl l ==> t_l

fun count0 [] = 0 |
    count0 (h_l::t_l) =
 if h_l=0
 then 1+(count0 t_l)
 else count0 t_l
```

Thence the comparison with the head of the list against `0` is replaced with two new patterned cases:

```
CHECK 6:
 CONVERT if <var>=<const> TO PATTERN
fun count0 [] = 0 |
    count0 (h_l::t_l) =
 if h_l=0
 then 1+(count0 t_l)
```

```
 else count0 t_l ==>

fun count0 [] = 0 |
    count0 (0::t_l) = 1+(count0 t_l) |
    count0 (h_l::t_l) = count0 t_l
```

Last the unused list head in the final case's pattern is replaced with the wildcard pattern:

```
CHECK 4:
 REMOVE UNUSED VARIABLE IN PATTERN
h_l ==> _

fun count0 [] = 0 |
    count0 (0::t_l) = 1+(count0 t_l) |
    count0 (_::t_l) = count0 t_l
```

Note that the unused variable was introduced by the checker itself.

7 Evaluation

The style checker was developed for use with the Ceilidh automatic assessment system for SML. In Ceilidh, students submit answers to exercises which are assessed empirically by comparing their outputs with those of model answers on standard test data. The original Ceilidh[1] for C also included simple style checking, primarily for lexical and syntactic style.

For SML, students supply type specifications for their functions for comparison with the specimen answers' types. The style checker is used to check for style mistakes. Students may also use the style checker independently to get style comments in the form discussed above.

Ceilidh for SML was first prototyped without the style checker with a first year class who had already studied Pascal for one term. Subsequently, the style checker was used independently to analyse the students' submissions. Note that at this stage checks for wildcard, and `hd` and `tl` use had not been incorporated into the checker:

```
check                      occurrences
--------------------------------
partial application     395 54.8%
boolean comparison      125 17.3%
var/const comp in if     80 11.1%
boolean if result        68  9.4%
boolean if condition     52  7.2%
--------------------------------
total occurrences       720 99.8%
```

Only the first positive check on each function was counted to avoid counting checks detected as a result of program transformation of earlier checks.

Partial application is a new concept for imperative programmers: the high frequency of positive checks for misuse (54.8%) is not surprising. However, explicit boolean comparison and results from conditional constructs are discouraged in imperative programming. Their presence in functional programs (26.7% of positive checks) corresponds to the above suggestion of more general novice confusion as to the status of boolean values. Positive compositional checks for poor pattern usage (18.3%) are least frequent suggesting student understanding and acceptance of this style of programming.

8 Conclusions

The style checker has been integrated into Ceilidh for SML, which is now used regularly at Heriot-Watt University to support SML teaching. Anecdotally, students seem to find style checks confusing initially but soon learn to satisfy them. Students may submit exercises as often as they like within a given period. A few seem to treat Ceilidh as if it were a game, seeking to maximise their marks by frequent submissions with minor, poorly thought out changes. Most, however, develop a functional programming style which satisfies the above requirements. A more substantial analysis of style checks of student programs is planned.

We intend to develop additional semantic style rules for functional programming, in particular for the use of higher order functions, for compositional processing of user defined concrete data types and to simplify non-exclusive function cases. The style checker will then be extended to include these new rules.

Finally, the style rules are "hard wired" in the current style checker. It would be interesting to adopt an existing program transformation system to work with style rules.

9 Acknowledgements

This work was supported by the UK Higher Education Funding Councils' Teaching and Learning Technology Programme as part of the Ceilidh automatic program assessment project. I would like to thank Sandra Foubister for integrating the style checker into Ceilidh for SML. I would also like to thank her and Diana Bental for monitoring its use by students.

I would like to thank Lachlan MacKinnon, Peter King and the anonymous referees for their constructive comments.

Ceilidh for SML is distributed as part of the Ceilidh automatic program assessment system by the Department of Computer Science, University of Nottingham, England at no charge:

```
http://www.cs.nott.ac.uk:80/
       ~ceilidh/ceilidh/
```

References

[1] S. Benford et al. Experience using the Ceilidh system. In *Proceedings of 1st All-Ireland Conference on the Teaching of Computing, Dublin City University, in Monitor: newsletter of the Computers in Teaching Initiative Centre for Computing*, number 4, pages 32–35, Sept 1993.

[2] R. E. Berry and B. A. E. Meekings. Style analysis of C programs. *Communications of the ACM*, 28(1):80–88, Jan 1985.

[3] E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[4] G. Dromey. *Program derivation: the development of programs from specifications*. Addison-Wesley, 1989.

[5] G. Dromey. A model for software product quality. *IEEE Transacations on Software Engineering*, 21(2):146–162, Feb 1995.

[6] W. Gillett. An interactive program advising system. *SIGCSE/SIGCUE Joint Symposium on Computer Science Education*, 8/2(1):335–341, Feb 1976.

[7] G. Hannemyr. letter. *SIGPLAN*, 18(4):16–18, April 1983.

[8] R. Harrison. The use of functional languages in teaching computer science. *Journal of Functional programming*, 3(1):67–75, Jan 1993.

[9] W. Harrison and C. R. Cook. A note on the Berry-Meekings style metric. *Communications of the ACM*, 29(2):123–125, Feb 1986.

[10] J. W. Howatt. On criteria for grading student programs. *SIGCSE*, 20(3):3–7, Sept 1994.

[11] *Software product evaluation - quality characteristics and guidelines for their use*. ISO/IEC Standard ISO-9126, 1991.

[12] B. W. Kernighan and P. J. Plauger. *The elements of programming style*. McGraw-Hill, 1978.

[13] A. Lake and C. Cook. STYLE: an automated program style analyzer for Pascal. *SIGCSE*, 22(3):29–33, Sept 1990.

[14] H. F. Ledgard, J. F. Hueras, and P. A. Nagin. *Pascal with style: programming proverbs*. Hayden, 1979.

[15] D. Marca. Some Pascal style guidelines. *SIGPLAN*, 16(4):70–80, April 1981.

[16] B. A. E. Meekings. Style analysis of Pascal programs. *SIGPLAN*, 18(9):45–54, Sept 1983.

[17] N. E. Miller and C. G. Peterson. A method for evaluating student written computer programs in an undergraduate computer science programming language course. *SIGCSE*, 12(4):9–17, Dec 1980.

[18] D. Olson. The reliability of analytic and holistic methods in rating students' computer programs. *19th SIGCSE Technical Sympoisium on Computer Science Education*, 20(1):293–298, Feb 1988.

[19] K. A. Redish and W. F. Smyth. Program style analysis: a natural by-product of program compilation. *Communications of the ACM*, 29(2):126–133, Feb 1986.

[20] M. J. Rees. Automatic assessment aids for Pascal programs. *SIGPLAN*, 17(10):33–42, Oct 1980.

[21] S. K. Robinson and I. S. Torsun. The automatic measurement of the relative merits of student programs. *SIGPLAN*, 12(4):80–93, April 1977.

[22] S. S. Robinson and M. L. Soffa. An instructional aid for student programs. *11th SIGCSE Technical Symposium on Computer Science Education*, 12(1):118–129, Feb 1980.

[23] D. Rosenthal. letter. *SIGPLAN*, 18(3):4–5, March 1983.

[24] R. D. Tennent. *Semantics of programming languages*. Prentice-Hall, 1991.

[25] P. B. V. Verth and A. Ralston. A system for the automatic grading of programming style. In *Proceedings of 5th National Conference on Educational Computing, IEEE, Baltimore, MD*, pages 208–213, June 1983.