Dysfunctional Programming: Teaching Programming using Formal Methods to Non-Computer Science Majors

Michael Barnett
Laboratory for Applied Logic
Department of Computer Science
University of Idaho
Moscow, Idaho 83844-1010
mbarnett@cs.uidaho.edu

Phillip Windley *
Department of Computer Science
Brigham Young University
Provo, Utah 84602
windley@cs.byu.edu

November 24, 1993

Corresponding author: Michael Barnett
Running head: Dysfunctional Programming

Abstract

In the fall of 1992, we taught an introductory programming course to a class of non-Computer Science majors. We followed an approach based on formal specifications, with an informal translation to Pascal as an implementation language. This anecdotal paper reports on our experiences and ideas for future attempts.

1 Introduction

It is our belief that students are poorly served by the traditional computer science curriculum. In this, we are hardly alone, for example, see [4, 5, 6, 11]. Computer Science is getting

*This work was performed while the author was at the University of Idaho.

more complex, not less; thus, the curriculum must teach students methods for coping with complexity.

Over the past 100 years or so, science and engineering disciplines have come to grips with this sort of problem by developing the mathematical tools necessary to analyze and design complex systems. The traditional science and engineering disciplines deal with continuous systems; the integral calculus is well suited for modeling and analysis in the continuous domain and is well developed. Computer systems are discrete; the mathematics for modeling and analysis in this domain is only beginning to be developed. Many different branches of mathematics and logic are being used to explore this area. Collectively, these mathematical tools are called *formal methods*.

The case for formal methods in computer science has been made many times; we do not repeat it here. We would like to see formal methods integrated into the curriculum, beginning with the first courses offered in the department. We believe that including formal methods in the curriculum has several benefits.

1. A curriculum based on formal methods provides students with a firm mathematical basis to undergird every topic and discipline.

2. It gives them an opportunity to use a mathematics that is connected to their field (unlike calculus and most discrete math courses).

3. Formal methods are independent from any particular programming notation. Coding a program is the last step in a series of refinements; by then a solution to the problem exists and can be implemented in any particular notation. This better prepares students for a discipline that is constantly changing.

To test our beliefs, we taught an experimental section of the introductory computer science class offered at the University of Idaho. Although the course is taken primarily by students from other disciplines, minors and majors in Computer Science do take it if they do not have prior programming experience. Our intent was not to explore the effect of formal methods on non-majors, but rather to see how introductory students irrespective of their discipline fared in a formal methods approach. However, the presence of many students outside the major became an overriding issue and thus the topic of this paper.

2 The Text

We used the textbook The Programming Process by Latham, Bush, and Cottam [10]. This text is a case-study based approach using formal specifications (in the specification language VDM [9]). Case-studies are presented as "real-world" problems, at least to the degree that can be accomplished in a classroom setting. We followed the book fairly closely during the semester; this article is, in some senses, a review of the book and the authors's particular methodology as well as of formal methods in general.

The two components of the course, case-studies and formal methods, are orthogonal: it is beyond the scope of this paper to discuss the reasons for their combination. Lack of space prevents us from exploring all of the arguments for a case study methodology, for an extended discussion see [2].

The aims of the book are given in the introduction [10, p. 4]:

> **What is this book about?**
>
> This book is not obsessed with teaching you all the details of any programming

> language. We want to teach you the *process* of programming. That includes how to know which program to write, as well as how to write it so that it does the correct job as soon as it is finished. If all we did was teach you the details of a particular programming language, then you'd only know how to *read* programs, and only in that language.

The methodology is given as a refinement sequence of four steps: [10, p. 5]:

Requirements Analysis This is the phase where at least some of the ambiguities in the problem description are resolved, using an informal process between the "customer" and the "programmer".

Formal Specification A formal specification is a mathematical statement of the properties that a solution to the problem should possess. It is intended to be as precise as possible and serves as the reference for the next two steps.

Program Design While the specification describes the results desired from the program, the design describes how the program brings about the results.

Program Implementation Once a design is complete, it is translated into a Pascal program.

Of course, feedback from some later step leads to changes in previous steps; we often went through several iterations before arriving at a finished implementation.

We spent the most time on specifications. Because of our preference for formal methods, we liked the book's reliance on mathematical notation. However, this required the students to learn VDM; because most of them knew very little math, teaching the concepts behind the specification language took up much of the semester. We felt the time was well spent, since requiring the specification to be in a formal language ensures that any ambiguities in the original requirements are completely removed.

Translating the specification into a program design was an informal process of creating an algorithm that appeared to match the specification. We tried to give design hints of the sort of algorithmic structures different specifications give rise to. For example, a specification which refers to a sequence will lead to an algorithm with some form of iteration in it. The designs were detailed enough for the translation to Pascal to be almost entirely a word by word transformation.

The course was not about teaching VDM; we used the structure of VDM specifications, and very simple logical formulas, quantification, mathematical statements, and simple data types (besides the scalar types we used only sets and sequences). A VDM specification comprises four parts: a name, an external interface (`ext`) listing the input variables (`rd` for read-only variables) and the output variables (`wr` for read/write variables), a precondition (`pre`) on the input variables, and a postcondition (`post`) relating the input variables to the output variables. For example, Figure 1 shows a specification for an operation `modify-seq` that modifies a sequence of integers, given a sequence of booleans. The precondition requires the lengths of the two sequences to be the same. The postcondition states that each integer in the sequence is to be incremented if the corresponding boolean value is **true**, and to be decremented if the corresponding boolean value is **false**. The notation $\overleftarrow{\mathtt{x}}$ refers to the value of `x` before the operation; the other option is to introduce a free variable in the precondition in order to refer to it in the postcondition.

```
modify-seq
ext   rd which-way : sequence of 𝔹
      wr x : sequence of ℤ
pre   len(which-way) = len(x)
post  for all i, 1 ≤ i ≤ len(x)
          if which-way[i] then x[i] = x↼[i] + 1 else x[i] = x↼[i] − 1
```

Figure 1: Specification for `modify-seq`.

3 The Students

There were eight sections of the course, of which ours was the only one not using the traditional syntax-driven approach. We began the semester with 34 students and ended with 22 students. This attrition rate was not excessive in comparison with other sections of the same course that were taught. Some of the students switched into other sections after hearing from friends how much easier the other sections were—a problem of introducing rigor in a piecemeal fashion.

Seven of the students went on to take the next computer science course the following semester: we identify them as either majors or minors. The rest of the students came from colleges and departments spread across the university, but with a majority from other engineering departments and business students. For those students, this course is a requirement for their degrees.

The final grades followed a fairly normal distribution: 4 A's, 4 B's, 9 C's, 3 D's and 2 F's. The students we believe to be minors or majors did slightly better: they accounted for 2 A's, 2 B's and 3 C's.

(a)	(b)

`square`
`ext   rd x : ℝ`
`      wr y : ℝ`
`pre   true`
`post  y = x * x`
	`square-root`
	`ext   rd x : ℝ`
	`      wr y : ℝ`
	`pre   true`
	`post  x = y * y`

Figure 2: Specifications for `square` and `square-root`.

4 The Semester

During the semester, we had several goals and many frustrations. This section outlines, in anecdotal form, the most important of those.

4.1 The separation of "what" from "how"

Specifications precisely describe the properties of a solution, without getting bogged down in the details of how a solution is to be implemented. This was the single most important point we hoped students would learn from the course. We wanted them to realize that unambiguous specifications were a crucial part of computer science. Since a large number of the students were from the College of Business, we used the business metaphor of specifications as contracts to bring this point across.

As an example of the difference between a specification and an implementation, we used the two functions `square` and `square-root`. Their specifications are shown in Figure 2. The simplicity of `square`, shown in Figure 2(a), makes it hard to point out a difference in the *what* from the *how*. Its implementation corresponds directly to the postcondition in the specification. Compare, however, the specification for `square-root` in Figure 2(b). While it is easy to specify the relation between x and y, it is no longer possible to implement the solution by animating (executing) the specification.

As the semester progressed, their performance on tests convinced us that the majority of students did learn to distinguish between *what* a computer program should accomplish from *how* it accomplishes it.

4.2 The Tower of Babel

Every stage of the programming process had a separate language for its expression: English for the requirements, VDM for the specification stage, a small pseudo-code language for the design stage, and Pascal for the implementation language. Throughout the semester, our students displayed confusion as to which elements belonged to which language. When asked for a specification, they wrote syntax that could only be part of a design. For designs, they wrote Pascal programs—interspersed with specification logic.

This confusion is not necessarily a bad thing; learning new concepts often entails a period when different levels and methods are mixed in inappropriate ways. Were the course to be linked with later courses in the curriculum, we believe students would eventually sort out the different languages and the levels to which they are appropriate.

It is harder to justify the confusion for students outside the major; for them, the course is supposed to be a self-contained introduction to computer science. Our conclusion is that three separate levels was too much. The important concepts to get across are the importance of algorithmic thought and precise descriptions. While it is necessary to teach students that different tools are appropriate for different purposes, the number of levels and the distance between them made it difficult to get this across without losing the students in the details.

Being lost in the details was easy for our students; the book was written for British students and generally assumed a much higher level of mathematical ability than our students demonstrated. Our impression is that British high schools produce students with more mathematical maturity than American high schools do. A book directed at American students would be of benefit for both majors and non-majors. For majors, the course could be extended over two semesters, and cover the mathematical material (propositional and predicate logic, sets, functions, and relations) in a more comprehensive manner, perhaps to the extent of replacing the discrete mathematics course our majors are now required to take.

Another option is to reduce the distance between the different levels by using a language that can function both as a specification language and an implementation language. For instance, a functional language augmented with the simple kind of logical language we used in the course would allow a refinement-based approach that does not require students to master a completely different syntax. More attention could be given to transformations that improve efficiency (the most important of which are the transformations that change an unexecutable logical statement into an algorithmic one).

4.3 To Func or not to Func

Once the programming process has been split into separate layers, care must be given to the way in which they are connected. The transition between two layers should be as well-justified as possible; it should provide guidelines for the beginning programmer to follow as they develop their own programs. Without this, the risk is that the examples done in class look very beautiful and elegant, but the student finds only dead ends and frustration when trying the same approach on their own. In this section, we use an example from the text to illustrate these issues.

While it may not be necessary, many specifications are naturally written in terms of functional abstractions. We present the program requirement from Chapter 4 in full [10, p. 85]:

> A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed. The words in the telegram is delimited by the word ZZZZ. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words ZZZZ and STOP are not chargeable and words of more than 12 letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word.

Leaving aside the ambiguities which are removed in the ensuing pages, the specification for this problem decomposes the problem into three separate problems. The top level specification is shown in Figure 3. It states that the operation is a function from `Blocks` to `Lines`. `Blocks` are sequences of `Block` and `Lines` are sequences of `Line`. Both `Block` and `Line` are of type sequence of character; VDM allows a type to specify also the length of a sequence. Thus, for the former, the sequence is of length 150 (the predetermined size in the requirement), while for the latter, it is of length 80 (for the "neat" listing).

The operation is decomposed into three separate functions, each of which takes as input

```
TELEGRAM-ANALYSIS
ext   rd tel : Blocks
      wr rep : Lines
pre   abs-Blocks(tel) is defined
post  rep = rep-Reports(analyse(abs-Blocks(tel)))
```

Figure 3: Specification for `TELEGRAM-ANALYSIS`

a sequence of indeterminate length. Each of the functions is responsible for transforming its input sequence into a sequence of elements used for input to the next function. The declarations for the functions are:

```
abs-Blocks   : (bs : Blocks)        ⟶  (tels : Telegrams)
analyze      : (tels : Telegrams)   ⟶  (reps : Reports)
rep-Reports  : (reps : Reports)     ⟶  (ls : Lines)
```

`Telegrams` are sequences of `Telegram`, and `Reports` are sequences of `Report`; the types for `Telegram` and `Report` are not important here.

The program is quite easy to understand. The stream of input is transformed by `abs-Blocks` into a stream of elements which correspond to the structure of a telegram. Then `analyze` produces a report from each telegram, and then `rep-Reports` converts each report into a format acceptable for output. This is a classic decomposition of many problems. It allows the central problem—the actual analysis—to be separated from the, possibly transient, details of input format and output format.

Previous chapters had used (sequential) text files as the simplest implementation of indeterminate length sequences, but they had all been simple enough that a single function sufficed to solve the problem. (While unconventional, we found that introducing files before arrays was actually one of the most positive feature of the semester.)

For this problem, remaining faithful to the specification created entirely new problems, none of which had a clean solution. Using files to implement sequences meant the implementation of `abs-Blocks` could read in a sequence of Blocks and write out a file of Telegrams for `analyze` to use as input. But the indeterminate length of the sequence of Blocks made it impossible to guarantee that there would be enough space for the file. (Were the course to be taught with a parallel language, `analyze` could be implemented as a concurrent process which could read the file while it is being written by `abs-Blocks`, but then that brings up the whole problem of synchronization.) Worse still, elements of Telegrams cannot be written to the file, but only a textual representation. Yet the whole point of the decomposition was to free the process of analyzing telegrams from considerations of concrete input format.

The solution was to expand the functions, and to design the program to use files only as the initial input and the final output. The first function had to read in enough input to create a Telegram that was then passed to the function which `analyze` applied to each element of the sequence it was applied to in the specification. Once the first telegram was processed, the function which `rep-Reports` called on each Report produced the final output corresponding to the telegram. This forces the program to operate in a lockstep fashion, which may or may not be desirable.

There are times when this type of restructuring is needed; it is a useful technique that programmers must know. However, the book makes no explanation of it at all, but just presents the design as an obvious refinement of the specification. This is not a criticism of the book; we are unaware of any work which provides a formal framework for such restructuring. Our solution was to present the options outlined above, but we desire a more satisfying approach that can be better justified.

4.4 The first programming language

Our attempt to convince students that the implementation language is of secondary importance ran counter to their desire to see their programs work, and their utter unfamiliarity with the use of a text editor/compiler/debugger system. For computer science majors the claim could be made that this is a hurdle which must be overcome; the sooner it is put into their path, the sooner they will gain the necessary proficiency. The argument becomes less convincing for non-majors (and indeed has been criticized even for computer science majors [1, 8]).

Functional languages have many benefits as a first programming language.

- The functional model of computation is based on the intuitive idea of substitution: equals can always be replaced by equals. Computations are expressed as a set of *rewrite rules*, by which expressions can be repeatedly simplified until a recognizable value results.

- Functional languages remove much of the syntactic overhead of imperative languages. Some implementations for imperative languages attempt to remove it by providing structure editors that "fill in the blanks"; this just makes it all the more mysterious to students who wonder about all of the extra statements that magically appear in their programs.

- The execution environment of functional languages is also important. Most come with a calculator style interface that allows the students to type expressions and definitions and receive values in return in a straight-forward manner. Compiling does not need to be mentioned, and editors can be introduced slowly.

- A functional model is neutral with respect to evaluation order. While imperative languages must add features to specify concurrency, functional languages support several methods of evaluation—including parallel evaluation. Since the model respects the principle of substitution, any part of a functional expression can be evaluated at any time.

- Perhaps most important is that functional languages hide much of the low-level implementation details: they have been called executable specifications [7, 12].

We find these arguments persuasive: we believe it would be better to use a functional language as the initial programming language both for majors and non-majors.

5 Conclusions

We could not possibly discuss the entire semester in one paper; instead we have tried to present the points which made the most impression on us. As with any impressionistic account, we may be guilty of bias which cannot be compensated for. Our conclusions take the form of questions to be answered by future work. Although our experience could be viewed as negative in some respects, we prefer to think of it as a vehicle for sharpening our concepts and plans. But—now that we have recovered from our shell-shock sufficiently—two questions are of primary importance:

1. Should the use of formal methods be restricted to majors, or is it appropriate for non-majors as well?

2. Can the functional and imperative approaches be reconciled? What is the overlap (if any) between the formal methods and the functional programming communities?

The functional approach seems better suited to a case study approach. Many "real-world" problems naturally break down via a functional decomposition. And, as noted in previous sections, there are many arguments for using a functional language as a first programming language. Yet, the trend among many in the formal methods community has been a Dijkstra-esque methodology, perhaps using [3]. Approaches such as [3] are elegant and mathematically pleasing. They have the advantage that most programmers in the world use imperative languages: starting them off with a formal approach is bound to be beneficial. Students can see the connection between a formal approach and later experience.

As to the particulars of the semester, some of the problems we discuss could have been glossed over, especially with an unsophisticated audience. We wished to reduce the amount of discontinuities, by providing bridges between separate pieces, rather than leaving the students to cope with them in an ad-hoc manner. Future research should provide a methodology for restructuring a specification based on a functional decomposition.

The computer science majors performed better in the course than the other students, but we are not comfortable in drawing many conclusions from this fact alone. They may have happened to be the better students overall, or it may be that our personal styles were more accustomed to teaching majors.

It is necessary to resist the expectation that a new methodology will have a revolutionary effect on the students. New methods do not work for all students. There will always be students that are not suited for the subject material in one way or another. The best students will succeed no matter what regime is used; our hope is to provide tools and guidelines that allow the average student to make progress. For these students, the current methods leave them with no guidance, but with a feeling that they are unable to perform the mystic incantations needed to become programmers.

Our enthusiasm for formal methods and belief in the need to use them in the introductory curriculum are undiminished. There is no alternative to the use of formal methods for ensuring the correctness of software, no matter whether one argues that all software should be correct, or only some safety-critical portion. It is important that formal method educators share their experiences—good and bad—and begin to distill a common base of experiences from which future, better curricula can be fashioned.

Acknowledgements

We would like to thank Vicki Almstrum and Chris Lengauer for their helpful comments. Thanks also go to the students who participated in the experiment, especially since they didn't have a choice.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall Inc., 1988.

[2] M.J. Clancy and M.C. Linn. *Designing Pascal Solutions: A Case Study Approach.* W. H. Freeman, 1992.

[3] E. Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs.* Springer-Verlag, 1990.

[4] E.W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1397–1414, Dec. 1989.

[5] D. Gries. Influences (or lack therof) of formalism in teaching programming and software engineering. In E.W. Dijkstra, editor, *Formal Development of Programs and Proofs*, chapter 18, pages 229–236. Addison-Wesley, 1990.

[6] D. Gries. Teaching calculation and discrimination: A more effective curriculum. *Communications of the ACM*, 34(3):45–55, Mar. 1991.

[7] P. Henderson. Functional programming, formal specification and rapid prototyping. *IEEE Transactions on Software Engineering*, SE-12(2):241–250, 1986.

[8] J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17–42. Addison-Wesley, 1990.

[9] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1990.

[10] J.T. Latham, V.J. Bush, and I.D. Cottam. *The Programming Process: An Introduction using VDM and Pascal.* Addison-Wesley, 1990.

[11] A.B. Tucker, B.H. Barnes, et al. Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force. IEEE Computer Society Press, 1990.

[12] D.A. Turner. Functional programs as executable specifications. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice-Hall Inc., 1985.