Introductory Programming Course: From Classics to Formal Methods

Juan Manuel Gutiérrez Cárdenas *
and
Ian Douglas Sanders †

Abstract—**Introductory courses in CS, especially those aimed at introductory programming concepts or fundamental concepts in CS, represent the core courses which give the student specific insight into what the body of knowledge of Computer Science entails. Due to this great importance the curriculum planning of these courses should be undertaken with special care. It is, however, a matter of some concern that many of these introductory courses are deficient in terms of a formal algorithmic approach to programming. This leads to a situation where undergraduate students base their algorithms and programs on "trial and error" and are content if their code "runs" rather than proving that the code they produce will always work correctly. In the longer term, we believe, this situation results in a great amount of low quality software produced by CS graduates.**

It is because of these problems that we argue that the lecturing of formal methods and correctness proofs must be taught as soon as possible in a student's CS studies. This paper presents a proposal that is aimed at introducing the student to a more formal algorithm and program development process, so that he/she becomes able to produce correct efficient software from the beginning of his/her programming career. We also feel that such an approach will give students a much better idea of the nature of CS than many typical introductory courses do.

Keywords: Formal Specifications, Correctness Proofs, Algorithm Complexity, Introductory Courses

1 Introduction

In many cases a student completing a course called "Introductory Programming Concepts" (or some similar name) gets the impression that the algorithms developed and presented by the lecturer are arrived at by some "magical" means. The students do not see that the algorithms are developed by some logical process and they are seldom shown that the algorithm is, in fact, a correct solution to the problem being considered. The students are implicitly asked to trust their lecturer's expertise and experience. This limited approach is then reinforced when the students are asked to develop their own programs without any instruction on how to do so. The students often produce an incorrect or non proven algorithm which they then transform into the source code of a (hopefully) "working" program. This process often results in the students erroneously thinking that if a program is correctly executed, compiled or interpreted, or maybe if it does not have any errors during a debugging process, then the algorithm is also correct. Often nothing is done to convince the students that their solution may not actually be correct in all instances. An optimistic way to deal with this issue would be to ignore these bad habits as the student progresses through his/her career in the hope that they will come to understand that more is required in developing *correct* code. Unfortunately we observe that this bad approach continues into graduate levels resulting in the quality of software produced by CS graduates being poor. An additional issue that we are concerned about is that students do not reflect on the algorithms that they have developed and do not consider issues like time and space complexity.

In this paper we argue that proactive measures should be taken to teach students that there is more to developing correct working code than simply writing and rewriting code until the program runs "correctly" on the given test input (or some restricted input set). We believe that the students should be shown that some kind of model or formal specification of the problem based on the problem description is required to be sure that the correct problem is being tackled and that this should be followed by a formal proof of correctness after the development of the algorithm or program. The purpose of doing this would be to help the student learn to abstract and to get to the real essence of the problem that he/she wants to solve, and also to be able to prove by mathematical means that their solutions are correct. In addition, we would like students to be able to argue about the efficiency of their solutions. We believe that doing this will mean that the students are taught a good way of developing code and that the quality of programs/software that they produce will improve.

*Sociedad Peruana de Computación. Email: wits.gutierrez@gmail.com
†School of Computer Science, University of the Witwatersrand, Johannesburg, South Africa. Email: ian@cs.wits.ac.za

We feel that the approach taken by the University of the Witwatersrand (Wits) [14] meets our requirements – see details of their approach in Section 3 below. In this paper we suggest that a similar approach could be applied to the universities in our country (Perú). Adopting an approach similar to that at Wits would address the problems mentioned above and at the same time prepare the students better for their future careers as computer scientists.

2 Background

2.1 General motivation

The need for the teaching of formal methods, proofs and complexity analysis of algorithms, not only at an intermediate course level, but also in the beginning undergraduate level, has clearly been specified in [7] who argues for the need for a formal approach oriented to the construction of software and also discusses how to do correctness proofs in the developing process of software. Another interesting approach is the one made in [3] in which (as well as discussing a very similar method to the made in the introductory course of Fundamental Algorithmic Concepts [14]) it quotes that "Students should learn to examine every program they write, to be sure it is correct and efficient". There also exist proposals like [11] in which is mentioned, for example, a method called *Reverse Derivation*, which tries to be used as an alternative to the teaching of formal methods in the first year of the career of CS, this approach is based mainly in heuristic methods. One of the more important conclusions derived from the prior reference is the defense of a course based on formal methods, because even though it does need an adequate rigid academic approach, this does not imply that the student will decrease his creativity at the moment of making programs.

The detractors to the teaching of these topics tend to argue that teaching these techniques along with the lecturing of logic, mathematical proofs (direct, contradiction or induction to name a few) and of the teaching of asymptotic complexity analysis of algorithms will lead to an inadequate use of the assigned time for the introductory topics in CS. For solving these issues we can take into account the statement in [10] where it is mentioned that for a first course in CS the student could work on small well-known algorithms at the beginning in order for the student to re-inforce and practice the correctness techniques already learned and then later the student should be encouraged to develop his own algorithms and programs based on the proofs and methodologies learned.

The need for inclusion of certain topics in mathematics, specifically proofs and their applications to algorithms has been already defined in [2]. In addition, in that article there is a mention about some tips and strategies on how to approach this topic mainly in relation to tutorials. Some discussion about how to make an adequate algorithm analysis in contraposition to the synthesis of these is made in [8]. It should also be noted that this author suggests that the lecturers or instructors should personally be in charge of these courses; because they are the ones that must take in account that the student does not come with the necessary analytic capacities or even worse if they know them, mainly learned in the high school, are in the majority of cases erroneous.

2.2 The current situation in Perú

In our country the CS1 courses follow the *Functional First* approach. The main reason for adopting this approach is because it facilitates the teaching of specific topics, for example binary search trees, and because there is less emphasis on learning the syntax of a particular programming language (functional languages are seen as being more accessible due to their simpler semantics[9]). In addition, the functional first approach has been chosen with the goal of leveling the playing field in that all the students will at the same level in their first semester at university because the functional paradigm is not taught in high school. Note that these motivations are similar to those of the School of Computer Science at the University of the Witwatersrand who chose to teach Scheme in their first year course [13].

Unfortunately even though the functional approach is chosen because of its versatility there are very few introductory programming courses which focus on the teaching of good (formal) program development skills/habits and most beginning programmers still develop their programs in an intuitive fashion [9]. The students will not normally be concerned with establishing, in a formal way, the correctness of their solution method and/or their final program but will be content if their code works on some limited test cases. The courses that the students take often do not make any attempt to introduce the students to the need for or the skills of proving that their programs are correct. This situation is very common in universities in Perú.

Another situation which is very common, in Perú and elsewhere, is that many students and instructors will have encountered imperative programming before being introduced to the functional paradigm and thus will impose the imperative style onto the functional language. This reduces the chances of the functional paradigm being used as a basis of teaching a formal approach to programming.

Another factor which we believe detracts from the possibility of students learning to develop correct working code is that in many cases the structure of the first year CS curriculum seems to be badly defined. Topics such as computational logic, mathematical proofs and basic analysis of algorithms are either not included in the first year courses or if they are offered at all then the top-

ics are lectured jointly with a basic mathematics or discrete mathematics course. This means that the student does not see the relevance of these topics and does not know how to use the techniques as they do not see a real Computer Science application of the techniques learned in those split up courses. Again this situation applies in Perú and in other parts of the world.

2.3 Summary

There are strong arguments for teaching formal methods plus the tools/techniques for dealing with formal methods as part of a first year computer science course. In addition, it seems that students from Perú would benefit from a curriculum which includes such material and is based on the functional first approach to teaching. The proposal which appears in Section 3 below has been made taking these factors into account as well as considering the teaching of CS1 in different universities [5, 6] as is specified on the web page of the Formal Methods Europe [4]. The proposal is based largely on the course of Fundamental Algorithmic Concepts as presented in Sanders and Mueller [13] and described in detail in Sanders [14]. We chose to follow the approach established by [14] because of the clear and precise treatment that it gives, and also because it brings up a parallelism with other topics such as: mathematics, discrete structures and software engineering. Other factors which influenced the proposal are that the chosen approach for CS1 in our institution is based on the Functional First paradigm according to the schemata made by [1] and because the undergraduate students who come from the superior high schools of the country where it was originally applied have a lot of similarity with the social, economic and educational levels that we currently find in Perú.

3 The Proposed Methodology

3.1 Overview

This methodology that we are proposing for introduction in Perú has already been formulated in [12, 13] because of the explicit need to

1. introduce the students to an approach which would allow them to verify the development of their algorithms and programs
2. give a real understanding of the nature of CS – taking away from the erroneous concept that CS is programming and
3. provide the undergraduate student with a good and accurate introduction to the real core of studying CS.

The approach for developing correct efficient algorithms and programs as used by Wits (as presented in [14]) is composed of the following steps:

1. Problem Description: This is a draft of the problem to be solved, at this stage it does not need to be so accurate. This is an important issue because the student will be faced with vague requirements and asked to deal with imprecisely specified problems in his/her professional career.
2. Developing a formal specification of the problem: Here the "effect" of the computational procedure is considered without knowing the "how" [14]. There exist different kinds of notations and procedures to develop the formal specification. In the proposed methodology a simple mathematical notation has been used which is easily understood by the students.
3. Developing an algorithm to solve the problem: Here the conversion from the formal specification to the algorithm is made. As described in [14] often this conversion is direct – the formal specification leads straight to a functional description of an algorithm in the functional paradigm. Here the student will have seen the formality of the mathematical notation expressed in the former point and will have a clearer view of what is required than if only he/she is only presented with the algorithm or pseudocode.
4. Coding the algorithm: The use of any type of functional language is recommended for this phase of the process.
5. Reasoning about the algorithm or program: Here simple mathematical proofs are used to show whether the algorithm or program are correct or not.
6. Analysing the algorithm: Here asymptotic analysis of the performance the developed algorithm is done. To make this part more comfortable to the student, who is a beginner in this topic, only the worst and the best case in the algorithmic analysis are considered. The topic of average case complexity is left for coming courses.

3.2 Syllabic Requirements

In order to follow the process outlined above the students have to be taught some theory and thus it is necessary to include the following topics in the CS1 course:

1. Mathematical Proofs: Direct, Contradiction and Induction, these kinds of proofs have been studied by the students in courses in high school. If this is not the case then this material could be taught in a period of approximately two weeks.
2. Asymptotic Complexity: The student come to the career with a basic knowledge of functions and

graphs, thus the lecturing of this topic will be straightforward.

3. Functional Language: Our recommendation would be Scheme with its GUI Dr Scheme – mainly because of its attractive environment and also the ease of the program development.

3.3 Case Studies

In this section we highlight the steps mentioned before by means of some examples:

Case 1: A step function

1. Problem Description: Make a program that returns the value of 1 if the input is 0 or more and returns the value of 0 in any other case.
2. Formal Specification:

```
f(x)=1 if x>=0
f(x)=0 if x<0
```

3. Developing an algorithm

```
Step(x)
    If x >= 0
    Then
        1
    Else
        0
```

4. Coding

```
(define (step x)
  (if (>=x 0)
    1
    0))
```

5. Reasoning about the algorithm/program: It can be easily seen from the formal specification and our algorithm that the algorithm will only return 1 if the condition x>=0 holds. For the case of our program, in Scheme we have the use of conditionals which evaluate according to the following formula
`if <condition> <consequent> <alternative>`.
In the program the `if <condition>` would evaluate to #t in the case that the input element is greater than or equal to 0 and it will return the value specified in `<consequent>`, that is 1; in the other case it will evaluate to the value of #f and will perform the statement corresponding to `<alternative>` returning the value of 0.
6. Analysing the Algorithm: The time of execution of this algorithm is O(1) because it does not depend of the value of x. Only one comparison is made to determine if x>=0.

Case 2: Searching a binary search tree

1. Problem Description: Make a program that will allow for finding an element in a binary tree
2. Formal Specification: Let A be a binary tree A(V,E), we have then a value b to be searched, then Search(A,b)=True if $\exists$ v $\in$ V, such that v.key=b and Search (A,b)=False otherwise.
3. Developing the Algorithm: For this case we would be need a recursive structure – so that the algorithm looks for the element k in the root of each sub tree and in the case of not finding it, the search will continue in the right or left subtree respectively.

```
Search(A,b)
If b=root then
    True
Else
    If b<root then
        If the left subtree exists
        then
            Search (left_subtree, b)
        Else
            False
    Else
        If the right subtree exists
        then
            Search (right_subtree, b)
        Else
            False
```

4. Coding the algorithm

```
(define (search b tree)
(let
  ((root (car tree))
   (left (cadr tree))
   (right (caddr tree)))
  (if (= b root)
      #t
      (if (< b root)
          (if (null? left)
              #f
              (search b left))
            (if (null? right)
                #f
              (search k right)))))
```

5. Reasoning about the algorithm/program: There are two different cases for this algorithm. The first case is when the element

to be found is in the tree and the second case is when the element is not. Both cases should be proved to be able to claim that the algorithm/program returns the right answer. In this example the first case is proved by a proof by contradiction and the second case is proved by inductive proof. These are as follows below:
First Case: Proof by contradiction that the algorithm will return #t if the element b is in the tree
Let us assume that we have a tree T with n elements containing some node v such that v.key = b and that the algorithm returns #f for this tree.
For this algorithm to return #f, we would have to have either the case in which b<root.key or b>root.key and the algorithm would then search in either the left or right subtree until there were no more elements. This would happen recursively. The algorithm would only reach an empty subtree if there is no subtree along the path searched such that subtree_root.key = b. Such a node exists and so we have a contradiction. Thus if the element b is in the tree then the algorithm will return #t
Second Case: Proof by induction that the algorithm will return #f if the element b is not in the tree
Base Case: Consider any tree T of height 0. If root.key != b then the algorithm would search in either the right or left subtree. Both of these subtrees are empty so the algorithm will return #f.
Inductive Step: We assume that for any tree of height<=h-1 the algorithm will correctly return #f when it does not find the element in the tree (IH).
Now we will use the IH to determine that the algorithm holds for a tree of height h.
The algorithm first considers the element at the root of the tree. This will not equal b so the algorithm will search for b in either the left or right subtree T_l or T_r. Both subtrees are of height at most h-1 and b is not in either subtree. By the induction hypothesis whichever subtree is considered the algorithm will return #f and thus the algorithm will return the value of #f for a tree of height h composed of $T_l \cup root \cup T_r$.

6. Algorithm Analysis:
It is clearly seen that the best case would be when the element is in the root, O(1), and the worst case is according the height of the tree, O(h).

If we consider the solutions to these trivial problems we see an intermediate step between the problem definition and the development of the algorithm. This step is the formal specification of the problem based in mathematical notation and gives the student a problem description without ambiguities. In addition, in most of the cases, the transformation from a formal specification to the algorithm is simple and so is the conversion of the algorithm into a working program in a functional language. In many introductory programming courses this step is neglected but we believe it is an important step to helping the student produce code which does not just "work" but is actually correct and does what it is supposed to do. The step of actually arguing about the correctness of the program re-inforces this idea in the student. Following this approach means that the student proves his algorithms and programs correct in a more real and effective way, preparing him/herself for the proofs of systems on a much greater scale such as those that he/she will face in the next years of his career. The last step in the process would be the performance analysis of the algorithm – this is done in an accessible manner for the first year students by using concepts and theory which they already understand from high school mathematics and by not attempting to cover the more difficult topics such as the average case analysis or how to do an empirical analysis of an algorithm. In this way we could also accomplish the goal stated in [1] which specifies that in many instances the careers in CS are related only with programming, by employing this methodology we will be giving the student greater breadth in the discipline.

3.4 Assessment of the method

The approach described in Sections 3.1 and 3.3 above has been used in the School of Computer at Wits since 1998 as part of the revised first year course [13] that was put into place in that year. The lecturers involved in teaching the course believe that it is accomplishing what it is designed to do – to teach the students fundamentals of Computer Science, to dispel the idea that CS is only about programming, to give the students a solid grounding in CS. In addition, the lecturers involved believe that the formal approach taken results in the students producing better code in later years of study and once they go out to work in industry.

In early 2009 Wits CS embarked on a short study [15] to test their graduates' perceptions of what is taught in first year – in particular when considering the process described in Section 3.1 above. Twenty seven graduates working in industry or in graduate programmes in various parts of world completed a short survey. The results of the survey showed that the Wits graduates feel that the fundamental concepts (logic, proof techniques, etc.) which are taught in the first year course are important for computer scientists, that the process for developing correct efficient code is important and that the process

leads to the development of better code.

4 Conclusions

Many introductory computer science courses focus quite intensively on programming. The students are, however, not shown good ways of developing correct working code but rather are taught syntax and then left to figure out how to make their programs work by "brute force" methods. This leads to bad code as well as to the impression that programming is hard.

In Perú the functional paradigm has been chosen in order to make programming more accessible but the emphasis on teaching is still on syntax. This means that the students do not see the importance of developing good software. We believe it is of great importance to try to improve the mistakes committed during the lecturing of these introductory courses, taking into account the experience gained from other universities, so that we could improve the syllabic content of the courses that entail the core of CS in the first year or the undergraduate topics inside our institutions. In short, we feel that students can and should to be taught how to develop correct efficient code.

In this paper we have presented a proposal for a teaching methodology for introductory programming courses in Perú based on the teaching of formal methods, mathematical proofs and algorithmic complexity analysis along with the normal content. The main aim of the proposed approach is to start to instruct undergraduate students in the importance of developing correct working code based on a formal process rather than developing code by more ad hoc means. In addition, the methodology is also intended correct the common misconception among students that Computation is Programming. The approach we proposed is based on a method that has been used successfully at Wits in South Africa for over 10 years and we believe it will be successful in Perú as well.

References

[1] ACM and IEEE working group, Roberts E. and Engel, G. (editors), *Computing curricula 2001 – Final Draft*, December 2001.

[2] Armoni, M., On the role of proofs in a course on design and analysis of algorithms. *SIGCSE Bulletin*, 38, 4, pp 39–42, 2006.

[3] Fekete, A. 1993. Reasoning about programs: integrating verification and analysis of algorithms into the introductory programming course. In *Proceedings of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education*, Indianapolis, Indiana, United States, February 18 - 19, pp 198–202, 1993.

[4] Formal Methods Europe, 20 Feb 2007, `<http://www.fmeurope.org/>`

[5] Foundations of Computer Science, University of Cambridge, 24 Feb 2007, `<http://www.cl.cam.ac.uk/Teaching/1998/\\FoundsCS/>`

[6] Foundations of Computer Science, SUNY at Stony Brook ; Department of Computer Science, 24 Feb 2007 `<http://www.cs.sunysb.edu/~cse113/>`

[7] Gries, D., Teaching calculation and discrimination: a more effective curriculum. *Communications of the ACM*, 34, 3, pp 44-55, March 1991.

[8] Henderson, P. B., Anatomy of an introductory computer science course. In *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*, Cincinnati, Ohio, United States, February 06 - 07, pp 257–264, 1986.

[9] Lau, K., Bush, V. J., and Jinks, P. J., Towards an introductory formal programming course. In *Proceedings of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, Phoenix, Arizona, United States, March 10 - 12, pp 121–125. 1994.

[10] Marion, W., CS1: what should we be teaching?. *SIGCSE Bulletin*, 31, 4, pp 35–38. 1999.

[11] McLoughlin, H. and Hely, K., Teaching formal programming to first year computer science students. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, United States, February 15 - 17, pp 155–159. 1996.

[12] Mueller, C., Rock, S., and Sanders, I., An improved first year course taking into account third world students. In *Proceedings of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education*, Indianapolis, Indiana, United States, February 18 - 19, pp 213-217, 1993.

[13] Sanders, I. and Mueller, C., A fundamentals-based curriculum for first year computer science. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, Austin, Texas, United States, March 07 - 12, pp 227–231, 2000.

[14] Sanders, I. D., *Fundamental Algorithmic Concepts Course Notes*, School of Computer Science, University of the Witwatersrand, Johannesburg, 2003.

[15] Sanders, I. D., *An assessment of the formal aspects of the Wits first year course*, Technical Report CS-TR-2009-01, School of Computer Science, University of the Witwatersrand, 2009.