Introducing Formal Methods via Program Derivation

Dipak L. Chaudhari
Department of Computer Science and Engg.
Indian Institute of Technology, Bombay
Mumbai, India, 400076
dipakc@cse.iitb.ac.in

Om Damani
Department of Computer Science and Engg.
Indian Institute of Technology, Bombay
Mumbai, India, 400076
damani@cse.iitb.ac.in

ABSTRACT
Existing attempts towards including formal methods in introductory programming courses essentially focus on introducing program verification concepts and tools. When using the verification tools, there is no structured help available to the students in the actual task of implementing the program, except for the hints provided by the failed proof obligations. In contrast, in the Correct by Construction programming methodology, programs are systematically derived from their specifications. By restricting attention to program derivation, we have been able to identify a small core of formal method concepts that can easily be taught in the first two years of a computing curricula. Based on our learning from multiple years of paper and pencil based teaching, we have developed a programming assistant system that addresses several of the issues faced by the students in the manual program derivation. In this work, we discuss the program derivation methodology with a detailed example, discuss the difficulties faced by the students, and the teaching tool that we have developed to address those difficulties. The tool ensures that the most common students’ error of performing incorrect proofs does not happen. To help student manage the long derivations, a graphical user interface facilitates seamless navigation across the different stages of the derivation, and allows the user to backtrack, reevaluate the earlier choices, and experiment with different derivation strategies. Initial student response to the tool has been very enthusiastic. The tool enables the students to focus on the creative aspect of the program derivation process by taking care of the underlying details.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education; Computer science education; D.2.4 [Software Engineering]: Software/Program Verification; Correctness Proofs, Formal Methods, Programming by Contract

General Terms
Algorithms, Verification, Human Factors

Keywords
Correct by Construction Programming, Calculational Style of Programming

1. INTRODUCTION
In its final report [1], the ITiCSE 2000 Working Group on Formal Methods Education aspired to see the concepts of formal methods integrated seamlessly into the computing curriculum. Fifteen years later that aspiration still remains an aspiration. In our opinion, the major reason for this is the fact that the points of integration identified in the report, in Appendices C and E, come much later in the curriculum. By that time, the students are already used to the informal ways of developing programs and software and the old habits die hard. Ideally formal methods should be introduced as early as possible, particularly when students are just learning how to design programs [4].

Existing attempts in this direction focus on employing formal verification for teaching program correctness[12, 3, 11, 6, 10]. The Implement-and-Verify program development methodology involves an implementation phase followed by a separate verification phase. Although the failed proof obligations provide some hint, there is no structured help available to the students in the actual task of implementing the programs. Students often rely on ad-hoc use cases and informal reasoning to guess the program constructs.

In contrast with this, in the Calculational Style of Programming (CSoP) [5, 8], programs are systematically derived from their specifications. At every step in the derivation process, a partially derived program/formula is transformed into another form, by following certain heuristics. The derived programs are correct-by-construction since the correctness is implicit in the program and formula transformations employed during the derivation. Since the students see the program transformation strategy that led to the introduction of a particular programming construct, they understand why a particular programming construct was introduced at a particular point in a program.

Based on the four offerings of a program derivation elective course to sophomores, we have identified a small core of formal method concepts using which derivation of a large number of programs can be taught. We have also identified several difficulties faced by the students in using the method effectively (discussed in Section 4). To address these
difficulties, we have developed a tool called PDS (Program Derivation System - actual name anonymized), for deriving sequential programs from their formal specifications. To the best of our knowledge, no comparable tool exists. As discussed before, existing tools [12, 3, 11, 6] only ensure the correctness of the already implemented programs. Besides providing counter-examples, these tools provide limited help to the students in learning the program design techniques.

In PDS, we have tried to keep the notation and derivation style as close to the pen-and-paper style of derivation as possible. Our main emphasis has been on the usability in the class; in particular on being able to model and replay the ad-hoc interactions and iterations that usually occur during the manual program derivation by the students.

The organization of the rest of the paper is as follows. In Section 2, we discuss the core formal method concepts employed by us for teaching CSoP. In Section 3, we illustrate the methodology with a detailed example. In Section 4, we discuss our experiences from the years of teaching CSoP to sophomore students. In Section 5, PDS system and the students experience with it is presented. Section 6 concludes the paper.

2. CORE IDEAS

The advantage of teaching formal methods via CSoP is that the students can quickly write programs to solve non-trivial problems after being exposed to a small set of concepts. After starting with the sum of an array and the maximum element of an array, we quickly move to the binary search, fast exponentiation, and the maximum segment sum. After that we cover various other optimal array segment and search problems, and other similar problems like decomposing a number in a sum of two squares. Then we move on to array rearrangement problems such as array partitioning and sorting. In one offering, we were able to cover even more advanced problems such as the area of the largest square under a histogram. We wish to emphasize that all of it can be done using a small set of formal method concepts. Besides propositional logic, and the concepts of assertions and loop invariants, we only need a formal concept of quantified expressions and the rules for manipulating them.

For representing quantified expressions, we use the Eindhoven notation \( \langle OP : R.i : T.i \rangle \) [8], which, outside the formal methods community, is typically used only for quantified terms in arithmetic(\( \sum, \prod \)). Here \( OP \) (say, \( \sum \) or \( MAX \)) is the quantified version of a symmetric and associative binary operator \( op \) (say, + or \( max \)), \( i \) is a list of dummy/quantified variables, \( R.i \) is the Range - a boolean expression restricting the possible values that the dummies can take, and \( T.i \) is the Term - over which the underlying binary operator is repeatedly applied. For example, just the way \( \langle \sum i : 0 \leq i < N \wedge A[i] \rangle \%2 = 0 : A[i] + A[i] \rangle \) represents the sum of the square of the even elements of the array \( A \). \( MAXi : 0 \leq i < N \wedge A[i] \rangle \%2 = 1 : A[i] + A[i] \rangle \) represents the maximum of the square of the odd elements of the array \( A \). By using the same notation for all the quantified terms - including the logical quantifiers (\( \forall, \exists \)) - we can have generalized calculational rules[8].

While a large number of quantified expression manipulation rules are known [9, 2, 8], we find that for our purpose only three rules suffice: Range Split, Empty Range and, One Point Rule. The Range Split rule is most commonly used to form an inductive hypothesis: to show that the loop body maintains the loop invariant. The Empty Range and the One Point rules are used to evaluate an expression when either zero or exactly one dummy satisfies the range condition. The entire expression is evaluated in an inductive fashion by applying the Range Split, and the Empty Range or the One Point rule. Due to the lack of space, we do not present a detailed discussion of these rules but only illustrate them with the help of an example.

Beside these rules, the only non-trivial concept from propositional logic that we use is that of Distributivity and its adaptation for the Quantiﬁer Calculus. Just the way * distributes over +: \( x * (y + z) = x * y + x * z \), similarly \( \wedge \) (logical end) and \( \vee \) (logical or) distribute over each other: \( (P \wedge (Q \vee R)) = (P \wedge Q) \vee (Q \wedge R) \), and + distributes over max: \( x + (y \max z) = (x + y)\max(x + z) \). With this small set of core manipulation rules, we can teach derivation of a large number of problems.

Just the way ITiCSE Working Group on Formal Methods [1] viewed formal methods as the "calculus" of software engineering, we view rules for manipulation of Quantified Operators as the "calculus" of program derivation.

3. MOTIVATING EXAMPLE

In this section, we derive the well-known Maximum Segment Sum problem by following the calculational style of derivation. This exercise highlights the typical steps that are involved in a program derivation session. The natural language specification for this problem is:

Let \( A[0..N] \) be an array of integers. Compute the maximal sum of the elements of all segments of \( A \).

This problem is formally specified in a natural fashion as shown in Figure 1(a), where \( S \) is the required postcondition. To do inductive computation, we introduce a fresh variable \( n \) and rewrite postcondition \( R \) as \( P_0 \land P_1 \land n = N \) where \( P_0 \) and \( P_1 \) are given in Figure 1(b).

Now we can take \( P_0 \land P_1 \) as the loop invariant and \( \neg(n = N) \) as the loop condition. We observe that \( P_0 \) and \( P_1 \) can be established initially by \( n := 0 \). At this stage, we arrive at the program shown in Figure 1(c) as the solution for \( S \).

We investigate the inductive step \( n := n + 1 \). Now, if we want the loop invariant \( P_0 \) to be true after the assignment, then we need the assertion \( P_0(n := n + 1) \) before the assignment, where \( P_0 (n := n + 1) \) represents the formula obtained by replacing \( n \) with \( n + 1 \) in the body of \( P_0 \). That is, if we want \( \{ r = (MAXp, q : 0 \leq p \leq q \leq n : Sum.p.q) \} \) to be true after \( n := n + 1 \) is executed then \( \{ r = (MAXp, q : 0 \leq p \leq q \leq n + 1 : Sum.p.q) \} \) must be true before the assignment is executed. This is called necessary(or weakest) precondition \( np.(n := n + 1).P_0 \) w.r.t. the assignment.

Now we expect to modify \( r \) to some \( r' \) before incrementing \( n \), where \( r' \) is a metavariable. A metavariable is not a program variable - it just represents an unknown expression. Then, \( np.(r := r').(np.(n := n + 1).P_0) \) is the necessary precondition for the assignment \( r := r' \). To calculate \( r' \), we assume \( P_0 \), \( P_1 \), and \( n \not= N \) and simplify the consequent of this formula as shown in Figure 1(d). Every step in the calculation is associated with a hint justifying the step. For brevity, we skip the proof of preservation of the loop invariant \( P_1 \).

In step 15 in Figure 1(d), the quantified expression under consideration is neither easily computable nor easily expressed in terms of the existing program variables. We,
therefore, introduce a new variable $s$ and add a loop invariant $P_2 : s = (MAXp : 0 \leq p \leq n : Sum.p.n)$. Observe that $P_2$ can be established initially by $s := 0$, since the summation over an empty range equals 0. We now arrive at the program shown in Figure 1(e).

Program $S_1$ has been added to ensure that $P_2(n := n + 1)$ is a precondition of the assignment to $r$. For $S_1$, we envision the assignment $s := s'$. By following the same procedure as before, we can derive the value of $s'$ to be $(s + A[n])$ max 0. In computing $s'$, once again we use the Empty Range rule that the summation over an empty range returns 0. The final derived program is presented in Figure 1(f). This completes our derivation. Note the small number of concepts that were needed to derive an elegant solution to a non-trivial problem. We next discuss our experience teaching this methodology.

4. COURSE FEEDBACK AND TOOL SUPPORT FOR CSOP

The students’ interest in the methodology is reflected in the course feedback where we received 87% score in the last offering of the course. Following two comments exemplify the students’ excitement: A quite different approach to programming, very innovating and interesting too. Some really great insights, and, We learned many good things. I never thought that program could be derived. The experience was enriching. Despite the mostly positive feedback, we also realized that students were facing a number of difficulties in manually (without using any tool support) deriving the programs:

Common Difficulties:

(CD0) Difficulty in understanding formal logic: Used to informal reasoning, students make several mistakes in understanding and applying inference rules. Our experience is consistent with several issues reported in [13].

(CD1) Not checking transformation applicability conditions: Many of the program transformation rules have prerequisites that need to be checked. For example, + distributes over quantified $\max$. Students often forget to check such conditions.

(CD2) Long derivations: Compared to the guess and test approach, the calculational derivations are longer even for simple programs. Students get restless if the derivation runs too long, leading to more errors.

(CD3) Mistakes made during guessing: Manual derivations often involve small jumps where the unknown program expressions are simple enough to be guessed easily. Students often inadvertently take big steps during guessing, resulting in incorrect program expressions. For example, for program $S_1$ in Figure 1(e), many students make a jump and guess
the value of $s'$ to be $s + A[n]$.

(CD4) Forgetting to add bounds to the introduced variables: It is a general guideline to add bounds for a newly introduce variable, such as the bounds for $n$ in the maximum segment sum problem. Students often forget to add such bounds, and later in the derivation, when the bound constraints are needed, they have to backtrack and take the corrective actions.

(CD5) Forgetting to prove proof obligations: With their focus on unraveling the unknown program fragments, students many times forget to prove some of the proof obligations.

(CD6) Problem with organizing derivation: The derivation process is not always linear; it involves multiple iterations involving failed derivation attempts. Students often fail organize the derivation in cases where they need to go back and make some corrective changes. Unorganized derivation often leads to some missing proofs of correctness.

Based on the errors experienced during the multiple course offerings, we decided to develop tool support for teaching this methodology. We next outline the required functionality for the desired tool support. First and foremost, we must ensure correctness of all the steps involved in the derivation. The manual derivations occasionally employ informal reasoning. For example, the Step 8 in the Figure 1(d) implicitly uses the rule that $\land$ distributes over $\lor$. To ensure correctness, we need to have a unified framework to manage program as well as formula transformations. We must have a mechanism for dealing with the long derivations. In addition to automating the tasks involved, having the ability to organize the long proofs is vital. We also need to maintain history to provide backtracking and branching functionality. Finally, the user interface should allow seamless navigation across the derivation history.

5. PDS TOOL

We now discuss the PDS tool for the CSoP methodology. Its core components are implemented in the Scala programming language. Interactive user interface is provided in the form of a web application. PDS uses the Z3 prover as a backend theorem prover. But the students need not concern themselves at all with how the theorem prover works. The implementation details of the system have been published elsewhere (reference hidden for anonymity) and here we concern ourselves only with the use of the tool in classroom teaching.

5.1 Program Derivation Methodology

Students incrementally transform a formal specification into a fully derived program by applying predefined transformation rules called Derivation Tactics. For example, two of the tactics that we employed in derivation in Figure 1 are
to the user. Similarly, while transforming formulas, the user may want to focus on a subformula while keeping the rest of the formula unchanged.

This focusing on a subcomponent is achieved by applying the StepInTactic which displays the subcomponent under consideration along with the context and hides the rest of the program. User can then transform this subcomponent to a desired form and apply the StepOutTactic to bring the focus back to the whole program. The StepInTactic application can be nested any level deep. In Figure 3, the portions of the tree enclosed by rectangles correspond to the transformations performed on the focused subcomponent. Whenever user steps into a subcomponent, a new Frame is created to store the appropriate contextual information. The contextual information can then be used during the transformations of the subcomponent.

**Example:** Figure 4 shows a calculation that uses the “Focusing on subformula” functionality. Contextual information (assumptions) for the inner frame are displayed at the top of the frame.

### 5.3.1 Automating Formula Transformations

In the manual calculations, all the steps are kept small enough to be manually verified by the user. This is the main reason why the program derivations are long even for simple problems, and formal methods are hated by several students. With a tool support, however, we can afford to take large steps, as long as the readability is maintained. In general, small steps are good for readability. However, there are situations where certain calculation is not important from the derivation point of view. We would like to automate such calculations. We employ a backend theorem prover to perform required proofs. This makes the program calculations very flexible and greatly reduces the length of derivations. This helps with the observed errors CD2 and CD5.

For example, calculations that do not involve any metavariables are good candidates for full automation. In Section 2, we skipped the proof for the preservation of the invariant \( P_1 : 0 \leq n \leq N \). The proof obligation for invariance of \( P_1 \) does not involve any metavariable, and hence is not of interest from the derivation point of view. Students resist doing such proofs. We, however, still need to discharge them to ensure correctness.

The proof obligation for \( P_1 \) can be directly transformed to \( true \) by applying a VerifiedTransformation tactic that uses the backend theorem prover. In case the theorem prover fails to discharge the proof obligation or proves it invalid, we have to revert back to the step-by-step way of proving. A failed calculational proof often provides clues about how to proceed further with the derivation. This takes care of the errors CD1 and CD3.

Another example is the calculations involved in verifying the applicability conditions for some tactics. Consider the “Empty Range” tactic for the summation that transforms the formula \( (\sum i : false : T.i) \) to 0. If a student wants to apply this tactic directly to \( (\sum i : R.i : T.i) \), she first needs to show that \( R.i \equiv false \) (which may take several steps) and then apply the tactic. Other way is to directly apply the Empty Range Tactic to \( (\sum i : R.i : T.i) \) and the tool ensures that the \( R.i \) is unsatisfiable.

Automated formula transformations takes care of the most of the common logic related errors (CD0).

---

**Figure 4:** Calculation of initial assignment \((q, r := 0, x)\) to establish invariant \(0 \leq r \land q \cdot y + r = x\) while deriving Integer Division program (Set \(q, r\) to the quotient & remainder of the division of \(x\) by \(y\)).

---

\[
\begin{align*}
& a > 0 \land y > 0 \Rightarrow \exists t' \land q \cdot y + t = x \\
& \text{Step into consequent of the implication.} \\
& 0 \leq t' \land q \cdot y + t = x \\
& \text{Guessing expression values: } t' = x \\
& 0 \leq x \land q \cdot y + x = x \\
& \text{Replace formula by an equivalent formula} \\
& q \cdot y = 0 \\
& \text{Guessing expression values: } q' = 0 \\
& 0 \cdot y = 0 \\
& \text{Replace formula by an equivalent formula} \\
& true
\end{align*}
\]

\[
\begin{align*}
& x > 0 \land y > 0 \\
& \text{StepOutTactic} \\
& 0 \leq t' \land q \cdot y + t = x \\
& \text{Replace formula by an equivalent formula} \\
& true
\end{align*}
\]

---

### 5.2 Derivation History and Backtracking

**PDS** maintains the complete derivation history in the form a derivation tree. The user can also branch off from any point in the derivation to explore different derivation strategies. This helps take care of the errors resulting from CD4 and CD6.

Figure 3 shows a schematic representation of a derivation tree generated after a typical program derivation session. Node 1 is the starting node representing the specification and node 14 represents the final derived program. Node 6 and node 9 are the nodes where the user faces some difficulties with the derivation and decides not to carry out the derivation further and prefers to backtrack and branch out. The backtracking mechanism makes it easier for the user to try out different alternatives with least amount of rework. **PDS** user interface also makes it easy to navigate across different solutions.

### 5.3 Focusing on Subcomponents

A partially derived program at some intermediate stage in the program derivation may contain multiple unsynthesized subprograms. The user may want to focus her attention on the derivation of one of these unknown subprograms. Hence it is desirable that all the context information required for the derivation of the subprogram is extracted and presented to the user.
5.4 Evaluation

The tool became ready to be used by the students only towards the end of the last offering of our program derivation course. It received very enthusiastic response from the students. We did an anonymous survey to get specific feedback about the tool. There were a total of fourteen responses. Ten students felt that the use of the tool increased their confidence in the correctness of the derived program, while three did not feel so, and one student was unsure. Same pattern was observed for the question whether the tool simplifies or complicates the task of the derivation. To the question of how would they like to derive the programs in future, five said using the tool alone, six said that they would like to use the tool along with paper and pencil, and three students commented that they would not use the tool. Eight out of the fourteen students also felt that the tool should have been introduced right from the beginning of the semester, while three suggested introduction around the middle of the semester, and three students felt that the tool should not be introduced at all but they did not write any comments. Due to the anonymity of the survey, we are unable to determine why three students did not like the tool at all.

Overall, we are quite happy with the use of tool in the course. The biggest advantage was that the students could not submit incorrect derivation. They could only submit either correct or partially correct answers; since programs were correct by construction at all stages (although they may have been incomplete). We could look at the derivation history of partial submissions and identify the problems because of which they were stuck at a particular point. Students were happy about the fact that they knew that their solution was correct before making the submission. Note that this adds a completely new dimension to the concept of automatic grading of assignments [7]. We plan to use the tool right from the beginning of the next offering to understand its shortcomings in detail.

One unexpected downside of the introduction of the tool was the increase in ad-hocism in some of the derivations. In the class, we teach various derivation heuristics and the corresponding proof obligations, and students are supposed to follow them in the manual derivation. However, with the tool trying to automatically discharge proof obligations, some students make wild guesses about the required program constructs, resulting in very inelegant programs. For example, rather than deriving the value of s′ in Figure 1(f), many students introduce several if statements enumerating different cases involving positive and negative values of s and A[n]. In comparison, max operator in our derivation can be implemented using a single if. Note that these programs were inelegant compared to what is possible with the derivation methodology, and not compared to what is achieved in the standard guess and test methodology. Essentially, these students use the tool as a program verification system and not as a program derivation system. This in some sense bypass the argument we made in the introduction section as to why the program derivation and not the program verification should be used to introduce formal methods.

6. CONCLUSION

Instead of program verification, we have been using program derivation as the vehicle for introducing formal methods in the introductory programming classes. This has been done employing only a small core of formal method concepts. Based on our experience in teaching this method to several batches, we have identified a list of common errors made by the students while deriving the programs manually, and have developed a programming assistant to take care of these problems. The preliminary student response to the tool has been very positive. Based on the learnings from the first offering of the tool, we plan to further enhance the tool and deploy it right from the beginning of the next offering of the course.

Acknowledgements.

The authors would like to thank the anonymous referees for their helpful comments. The work of the first author was supported by the Tata Consultancy Services (TCS) Research Fellowship.

7. REFERENCES


