Deepening Students' Understanding of Algorithms: Effects of Problem Context and Feedback Regarding Algorithmic Abstraction

Leigh Ann Sudol

June 2011

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mark Stehlik, Co-Chair
Sharon Carver, Co-Chair
Ken Koedinger
Frank Pfenning
Carsten Schulte

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

In the study of computer science, programming is a fundamental skill, not just for the production of code, but for the practice of problem decomposition and algorithmic production. In introductory classes, students are expected to learn the skills to decompose problems into component parts, select appropriate algorithmic strategies, and compose those various strategies into a semantically and syntactically correct program. The challenge for students is understanding that programming is not simply about constructing individually correct lines of code, but how those lines work together to produce a coherent algorithm. Current technological approaches to developing pedagogical tools for supporting code production focus on either supporting students at the beginning of the process by prompting for small steps in the code writing process, or scaffolding the debugging process after the student produces compilable code. These approaches do not support deeper understanding of the algorithms that will transfer to future problem solving situations. The early assistance robs them of the opportunity to navigate the search space to retrieve the strategies appropriate for the particular problem being solved. Although post-compilation tools allow students to search for appropriate strategies and evaluate their results, the compilation prerequisite for their use requires that students spend a large portion of the debugging process without the help of the tool. The popular approach of applying context to the problem space could help support students during code production, but has not been rigourously evaluated.

This thesis takes an interdisciplinary approach which combines the application of code analysis tools from computer science and a pedagogical model from cognitive science. The goal of this approach is to deepen the understanding of algorithms through feedback in the appropriate problem solving stage. This allows for the development of a pedagogical tool that supports students at a point in the process that allows them both to gain practice in navigating the search space of problem solving strategies and provide algorithmic feedback pre-compilation. This dissertation includes the formulation and assessment of a model of student problem solving in a specific algorithmic domain, an analysis of the accuracy of the evaluation of student code within the system, and the evaluation of these variables on the learning and transfer on high school and college students taking introductory computer science courses. The learning and transfer will be evaluated through the application of automated feedback during the problem solving process, and compared against the most advanced of current approaches in computer science education. Additionally, I will experimentally test the common practice of adding context as a way of scaffolding student problem solving. The contribution to computer science education lies in the construction of a pedagogical IDE which evaluates code before compilation to produce semantic feedback for known problems. For the learning sciences, this work will show the impact of feedback regarding algorithmic abstraction in code production tasks, as well as inform the debate regarding the addition of context to problem solving situations.

Contents

List of Figures

List of Tables

1 Introduction

Computer science is an important part of literacy in the digital age. Across the United States the Computing in the Core movement[1] and the institution of a National Assessment of Educational Progress (NAEP) in technology[2] are bringing computer science into every child's expected academic program. In the introductory curriculum programming is the skill, and particular programming languages the syntactic and semantic structure, we require of students as they express their computer science knowledge[3].

Over the past 20 years the demand for students to gain early experience in professional level languages has influenced the introductory language choices. These languages, such as Java, C++, and C, have extraordinarily rigorous syntactical requirements and students are limited in the pedagogical tools and environments available during the learning process. These factors, combined with others, have led to a social consensus that computer science is hard, and an eventual decline in enrollments which is currently identified as a national security risk[4].

An important part of implementing an algorithm in a programming language is the decomposition of the problem into its component parts, selecting the appropriate strategies relevant to the problem, and recombining those strategies into a semantically correct algorithm[5]. The student must then write code for the algorithm to be compiled, tested, and run it in order to check their thought processes, and communicate the semantics to the computer. This process has been shown to be extraordinarily difficult for students, and an analysis of a student's solution to a problem can help illustrate why. Table 1 shows a problem given on the Advanced Placement Computer Science A exam[1] in 2007 and a response which received a middle range score. Although the AP exam requires a pencil and paper submission, you could equate this with a student's first attempt with a compiler or other tool.

The solution shown in Figure 1 was published as a part of workshop materials to help instruct teachers as to how student exams are scored. It shows code produced by a weak student who makes a series of semantic errors.

After constructing this solution, the next step in the student's process would be to fix any compile errors in order to progress to a compilable state where the program's semantics could be checked. Unfortunately for the student, the semantic errors such as the line of code "$if(sheets.get(n) > LargestScore)$" would be identified as a syntax error about an inappropriate l-value for a > sign. Depending on the student's misunderstanding, this syntax error may or may not clue them in to the fact that they need to access a particular part of the object retrieved by sheets.get(n). These syntax errors would need to be corrected before any semantic debugging could occur with the help of the system. By receiving feedback (syntax errors) designed for professionals with an understanding of the code they have produced, novices often focus on the low-level, local details surrounding the errors[6] and have extreme difficulty making productive edits[7]. This leaves the novices without explicit cues to look at the algorithmic semantics and develop a deeper understanding of the algorithmic components they are using.

[1]The AP Computer Science A exam is taken annually by approximately 15,000 students and its curriculum is designed to match a majority of the undergraduate CS curricula in the US.

Problem Description	Write the TestResults method highestScoringStudent, which returns the name of the student who recieved the highest score on the test represented by the parameter key. If there is more than one student with the highest score, the name of any one of these highest-scoring students may be returned. You may assume that the size of each answer sheet represented in the ArrayList sheets is equal to the size of the ArrayList key.
Goals for Student	In this problem the student needs to loop over the ArrayList named sheets from another part of the problem[2], involving determining the number of elements contained within the ArrayList and correctly accessing an element within the context of the loop. They need to determine the highest score through a comparison with a variable used to store the largest element. Finally they need to access the name of the highest scoring student and use a return statement to share the result. Because of the complexity of the data structures used in this problem that has several syntactical and semantic requirements.

Table 1: Example Problem With a Description of Student Goals

1.1 Current Technological Approaches

Current technological approaches to scaffold student algorithm production in code suffer from one or more of the following problems. The problem that impacts the most effective tools is that they are out-moded, the language that they offer tutoring in is no longer taught in introductory classes.. The lisp tutor[8] and other tutoring systems for prolog[9], pascal[10] and other languages have all demonstrated learning gains for students in the past, however they do not deal with modern languages and the syntactic complexities that they bring. Some recent systems attempt to automate the testing of student code through various representations to concretize the students' solution[11, 12, 13]. Unfortunately the automated testing tools only work on code which compiles. Finally, some tools prompt students for each individual line of code in a solution[14], which does help during the problem solving process, but again focuses the student to the individual lines of code, not the way that the code works together to produce an algorithm. The problem of requiring compilation or providing too much scaffolding up front does not support students' acquisition of the deeper knowledge required to transfer their programming skills to new algorithms or problem solving domains.

1.2 Using Context to Scaffold the Gap

A common pedagogical approach to helping students during the code writing and debugging process is the application of a context to the problem[15]. In addition to providing motivation for students to appreciate the problems by framing them in an interesting domain, such as digital media or robotics, the addition of context to the individual problems is hoped to provide a framework for reasoning for the student[**?**]. In an educational climate where solving real world problems is considered to be of highest value, the addition of problem context in computer science has the naive potential to both help students during the process and also

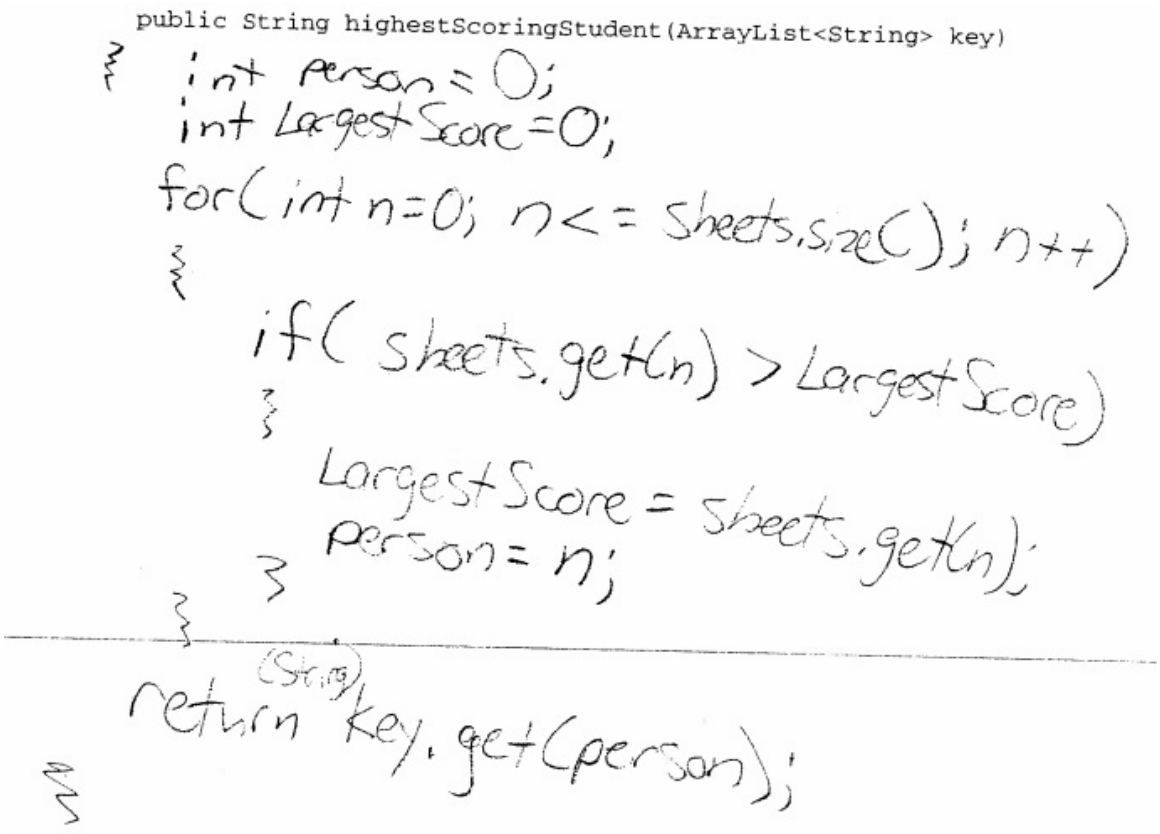

Figure 1: Components of Simple Array Algorithms

give them experience in real world situations[16]. No work has been found that tests this claim experimentally, and related work from other domains offers mixed results[17].

2 Theoretical Foundation

In order to address the need for balanced support during the student problem solving process, this thesis proposes a solution that comes from a technological contribution of evaluating student code pre-compilation using a combination of problem knowledge and static analysis tools, as well as a cognitive science contribution of feedback regarding algorithmic abstraction during the problem solving process. Although prompting the students for the next step in the process does help them during the supported activities, it does not help build a deep understanding of the decision making process employed in the construction of those steps[9]. In order to make those connections explicit, the Pedagogical Integrated Development Environment[3](PIDE) proposed in this thesis will test the impact of explicit feedback regarding the algorithmic semantics of students programs during the problem solving process.

[3]A pedagogical IDE is one that provides support for the novice computer scientist in ways that traditional IDEs do not. For example, the BlueJ IDE simplifies the buttons and menu commands as well as offers a visualization tool for object structures. The online practice system CodingBat offers specific problems with built in test cases for students to interact with.

2.1 The Benefit of Abstraction

There is a wealth and range of literature about abstraction in student reasoning and the correlations it has with other measures of student performance (e.g., code production)[**?**]. Many models have been proposed and evaluated for scoring the complexity of student responses to open ended questions[18, 19]. In computer science education, two popular models of student response complexity, and implied expertise, are the Block Model[20] and the Solo Taxonomy which was first proposed by Biggs and Collings as a generalizable model[21] and later adapted specifically to computer science[22, 23, 24]. Overall, these models and the research surrounding them indicate that as students progress from novice to expert they are better at identifying the relevant parts of an algorithm, and then connecting those parts through an understanding of their interactions. This becomes troublesome for the novices writing code. Ko and Meyers have shown that students not only have trouble understanding other peoples' code, but also code that they themselves have written[25].

Models of expertise in other domains also indicate that abstraction is an indicator of expertise in understanding complex systems. The seminal studies in expertise and abstraction were done with chess players[26], and recalling board states. Expert chess players were more accurate at remembering board states of playable games than novices. The experts did not memorize the locations, instead they chunked the board states in to relevant patterns of player behavior. Novices did not have smaller working memories than the experts, but were instead unable to build the complex mental representations of the chess board that allowed the experts to reproduce the board and have greater flexibility in their working memory when solving problems.

The correlational studies done in the computer science education literature are the theoretical foundation for offering explicit instruction in the form of feedback to students regarding algorithmic components and their relationship to each other.

2.2 Choosing the Right Domain

In order to select an appropriate domain for this work several factors were considered: common curricular practice, problem difficulty, number of subgoals or tasks in the problem and relevance to future learning/coursework activities. The choice to work with simple array algorithms is a departure from the loop-based problems explored in most of the prior literature[4][27]. Although problems involving arrays also involve loops, many of the early studies looked at looping algorithms where students would hand-enter the data at runtime, and the program was not required to maintain a data structure containing all of the entered data. As the languages changed, so too did the paradigms used in introductory programming. Larger data sets combined with increased modularity of programs in an object oriented language mean students rarely hand-enter data in modern classes, and make use of data structures earlier and for a broader range of problems. I make the distinction here as the problems chosen for these studies have the data pre-entered into the data structure for the student.

Thee specific problems were chosen to be difficult enough to require students to combine

[4]Many early studies in algorithm generation by novices made use of the "rainfall problem" requiring students to read in a list of values of rainfall amounts and calculate the average rainfall for the area.

several strategies to produce a result, but not so complex that the number of subgoals or tasks become prohibitive in the ability to construct the Pedagogical IDE to handle all cases. Finally these algorithms and the strategies they require are foundational for more complex algorithms often introduced in later lessons or courses. Additionally, because students are not looping over input data to perform actions, these algorithms are often the first time students encounter code complex enough that algorithmic abstraction is warranted.

2.3 Designing a model

An initial model of code production was built using a combination of similar models published by Elliot Soloway[5, 27] and expert knowledge. Starting with the ultimate goal of solving the problem presented, the student needs to choose which strategies are appropriate for the task at hand. Although the student does not need to select the strategies in any particular order, there are dependencies between the strategies as some variables may be necessary for multiple strategies. An example of this can be seen in an algorithm to compute the sum of the values stored in an array. The variable used to maintain the sum would be necessary both in the update of state variables and the return appropriate value strategy.

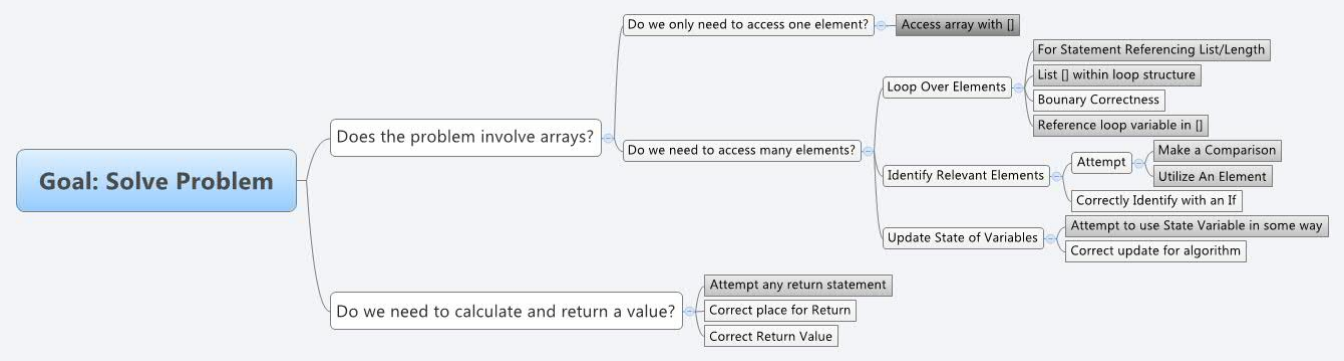

Figure 2: Components of Simple Array Algorithms

In the model shown in Figure 2 the student needs to choose from a set of goals for the code or method that they are writing. What is shown above is only a subset of those goals as they may apply to simple array algorithms, the domain chosen for these studies. The leaves of the decision tree (boxes all the way to the right) are the elements of the model that will be tested by the Pedagogical IDE. Depending upon which model constraints are satisfied by the student's program, the feedback will be targeted to either encourage them to utilize a current strategy, or refine their strategy to be correct. For example, if a student misses both of the components of the "Update State Variables" strategy in a program that requires it, they will receive feedback indicating that they need to utilize that particular strategy. If the student has attempted to use the state variables, but not correctly, they will receive feedback to help them with the correct implementation of that strategy as opposed to encouragement to use the strategy. The exact wording of the feedback will be tested and refined in the studies.

3 Research Questions

This thesis examines the hypotheses that:

(a)Feedback about semantic errors with regards to algorithmic abstraction will produce better performance in the writing of code by novices; and

(b)Contextual scaffolding impacts performance in the writing code by novices.

The studies proposed here focus on the problem-solving processes students engage in while producing code to solve simple array algorithms, and then debugging that code in order to achieve a correct solution. The variables of exposure to semantic feedback regarding algorithmic abstraction and the addition of contextual scaffolding, or not, to problems are evaluated through the three studies proposed in the next section.

4 Proposed Studies

The impact of instruction in abstraction and the addition of problem context on students' ability to produce code will be investigated and tested using a combination of quantitative and qualitative methods. The work proposed in this research plan investigates the impact of feedback regarding abstraction within the code production process. A Pedagogical Integrated Development Environment (PIDE) will be constructed to facilitate these studies.

This thesis offers contributions both in the learning and computer sciences. With respect to the role of abstraction in novices studying computer science, the existing literature is entirely correlational and is more focused on the assessment of student answers than on student learning. This thesis utilizes an experimental design integrating both qualitative and quantitative analyses to understand the impact of feedback regarding abstraction on student learning and performance. Additionally, prior work regarding algorithmic abstraction has focused on learner characteristics and assessment, while the PIDE proposed here will use explicit feedback in order to produce learning gains.

The technical contribution of this thesis is a novel approach to generating appropriate pedagogical feedback for novices engaged in a code production task as a part of their learning. The PIDE constructed for the studies involves code analysis in order to produce appropriate feedback. While many tutoring systems have been published for computer science, few include a methodology for analyzing student code prior to compilation to produce feedback that is different from the professional level tools available on the market. I take a novel approach of using a model-based grading system in order to determine the important characteristics of a correct solution, and categorize the type of feedback that students receive.

4.1 Study 1: Pilot Testing the PIDE and Model for Feedback Generation

The proposed model displayed in Figure 2 was developed using expert knowledge and intuition of the problem solving space. The goals of Study 1 are two-fold. First, we will validate the model through alignment with student thought processes during think-aloud protocols where they will solve the type of programming assignments found in Study 3. Secondly, the

feedback generated needs to be understandable to students and help them with the code production process. This study will collect the code produced at compile time by the students, the verbal think-aloud protocols given by the students as they are completing the problems and finally their ability to use the feedback produced by the system during the problem solving process.

The Domain

The exercises chosen for this intervention are based on array algorithms that students typically see during an introductory course[28]. Although the topic of array algorithms may be an advanced topic for a introductory course, it was chosen for the following reasons:

- The algorithm implementations, while brief, contain a number of conceptual chunks, thereby requiring abstraction or perceptual chunking to hold in working memory.
- Pilot studies using worked examples with simple looping programs yielded results at ceiling[**?**], indicating the need for increased difficulty.

Students will be asked to write short methods that make use of the following algorithms:

- Algorithms that involve a change to each element, requiring no selection such as sum all and increment.
- Algorithms that require a selection of elements, such as min/max search, count less than/greater than/property, sum select.
- Algorithms that require a selection of elements, with the potential for a return once the element is found, such as indexof.
- Transfer questions include algorithms with the additional goal of interacting with nearby array elements, such as isInOrder, nextTo.

The Task

During this study students will come to an office space or conference room to engage in a think-aloud protocol while solving problems from the tutoring sequence. Participants will first complete a pretest where they answer questions with no feedback. Then they will be given the tutoring sequence consisting of four problems delivered in a pedagogical IDE. After completing the tutoring sequence they will receive a post test. Students will be asked to think aloud during all parts of the process, and screen capture software will be used to record edits the students make on the screen between compiles.

The students will be randomized to one of four conditions: Abstraction Feedback No Context(AFN), Abstraction Feedback Context(AFC), Output Feedback No Context(OFN), Output Feedback Context(OFC). Table 4.1 shows two problem descriptions, one from the Context conditions and one from the No Context conditions. Aside from the problem description and name of the array variable being used there will be no difference between Context and No Context - specifically the feedback across those conditions will be the same.

The level of feedback provided is the other variable in this study. In order to provide students feedback a rubric grading system combined with technical implementation described in Section 4.3 will be used. Consider a solution to the problem proposed in Table 4.1 where

No Context	Context
Please complete the method findMinimum below. The method takes an array of numbers called myList as a parameter and returns the minimum value stored in the array.	Please complete the method leastExpensive. The method takes an array named prices containing the difference in price between a particular store's gallon of gas, and the mean price. The method should return the minimum value stored in the array.

Table 2: Example Code Production Problem

the student incorrectly initialized the variable that keeps track of the minimum value starting at an arbitrary value (0) instead of the first element in the list. This solution is given below:

```
public double findMinimum(double myList[]){
  double min = 0;
  for(int i=0; i<myList.length; i++){
    if(myList[i] < min)
      min = myList[i];
  }
  return min;
}
```

Assuming that the student was provided an example (edge case) where this would be a problem, the following are examples of the feedback the student might receive:

Output Based	Abstract Feedback
Student enters: [1, 5, -3, 2, 8] Student gets: 0	Your answer is incorrect. You did not find the correct minimum value, you may want to check your initialization.

Table 3: Example Feedback in Code Production Problems

In both conditions students will receive the same success conditions.

Anticipated Outcomes

As a result of this study I will be able to evaluate the following factors:

Model Refinement and Validation Working with students engaged in think aloud protocols during the problem solving process will be used to refine and validate the model offered in Figure 2 and used to generate the feedback in future iterations of the Pedagogical IDE.

Feedback Comprehension During the think alouds the wording of the feedback will be evaluated for comprehension and ability to direct students to the appropriate errors

in their code. It is anticipated that multiple iterations of feedback messages may be required during the think alouds.

Use of Contextual Scaffolding During Reasoning Think aloud transcripts will be coded for the use of contextual scaffolding during the reasoning process.

Special Cases

If during the initial think alouds students do not produce problem states that require feedback from all the model stages, additional think alouds will be conducted where students are given a pre-created buggy piece of code that will elicit the desired feedback. Although this will not help with the first goal of the study, it will ensure the evaluation and refinement of the error messages for all states of the model.

4.2 Study 2: Refinement of Code Evaluation Model

An important aspect of the Pedagogical IDE is its accuracy in diagnosing student problems. This study will focus on assessing and improving the accuracy of the part of the pedagogical IDE that creates a representation of the student's code and then generates feedback for the student.

The Data

The programs and attempts created by the students during the think alouds in Study 1 will be used as the dataset for the evaluation. The programs will then be rated by at least two human raters against the model. The human scores will be used as the target values for the PIDE.

Technical Evaluation

The pedagogical IDE will undergo an iterative process of refinement designed to increase the accuracy of the tests against the student data. In order to not overfit to the dataset a testing set will be separated from the data and used only for the final evaluation, not for the iterative refinement. In addition to correctness testing, the pedagogical IDE will also be evaluated against exhaustive JUnit testing to provide a comparative analysis of the types of feedback students would receive, and the percentage of errors that would be identified as syntactic and then semantic.

Preliminary Results

This approach to evaluating code was explored in a proof of concept study with an undergraduate researcher, Alexandra Johnson, in the spring of 2011[29]. Although the study used researcher-generated programs designed to elicit particular errors by the system, the system out-performed unit testing in its ability to detect and diagnose semantic errors within incorrect code.

4.3 Study 3: Learning Gains from Abstract Feedback

Study 3 will evaluate the impact of instruction on algorithmic abstraction and contextual scaffolding during the code production process. This study will also test the efficacy of the primary technical contribution: generating pedagogical feedback for novices using problem knowledge and abstract syntax trees.

The Task

While interacting with a web-based pedagogical integrated development environment (PIDE), students will complete small methods requiring that they implement simple array algorithms. Computer programming is an iterative task, and few novices are able to produce correct code on their first attempt. Within each iteration, students receive feedback from the compiler or the system about either compile or runtime errors. Students will be randomized to one of four conditions, varying either the feedback they receive or the application of contextual scaffolding to the problem statement. The four conditions are Abstraction Feedback No Context(AFN), Abstraction Feedback Context(AFC), Output Feedback No Context(OFN), Output Feedback Context(OFC). The details of the conditions are the same as described in Study 1.

Research Design

The effects of feedback mechanisms on a student's ability to solve problems and learning gains will be evaluated using qualitative data gathering from the tutor as well as qualitative methods from think aloud protocols. Figure 3 shows the planned research design and evaluation points.

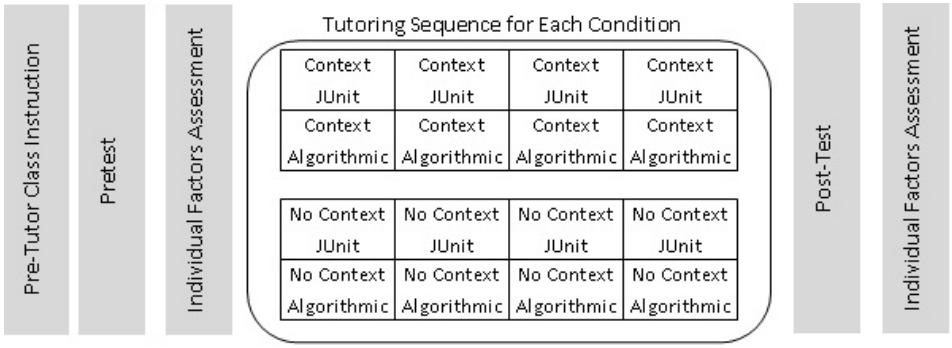

Figure 3: Study 2 Design: Students will complete at least 4 code production activities in each of the four conditions.

The participants will be students enrolled in a first or second semester computer science course. The pre-tutor data collection will include the same individual factors assessment used in Study 1. The pretest will consist of code writing questions without any feedback from the system[5]. Students will also answer some code comprehension questions as well

[5]This means that the student will not receive any compiler or testing feedback, their first submitted answer will be graded as is.

(potential transfer measure) involving algorithms both seen during the tutoring sequence and not seen.

Students will interact with the tutor, completing a series of code production activities over several questions. In the Non-Abstract condition, students will only receive example-based feedback (see table 4.1), while the students in the Abstract condition receive feedback about why their answer is incorrect.

After completing the tutoring portion, students will complete a post-instruction assessment. This assessment will include the same individual factors assessment from Study 1 and the pretest, as well as an assessment of computer science skills. The computer science skills assessment will include code writing questions which students will complete without any feedback from the system. The code writing questions will ask students to write code snippets seen during the instruction (direct assessment) as well as not seen(near transfer). The students will also receive code comprehension questions about snippets seen during the instruction (near transfer) and not seen (far transfer).

In addition to the pre and post test measures within-instruction data will be collected including initial code submissions, latency, and ability to resolve feedback messages. In addition students will be brought into the lab to complete think aloud protocols while interacting with the system to look for differences in the way that students respond to the feedback in the different conditions.

Expected Results

Based on the theoretical foundation the following results are anticipated:

Learning Gains All conditions are expected to show pre to post-test improvement on the direct measures of instruction. Students in the abstract condition are expected to perform better on the transfer measures as a result of the feedback during the study. Students in the contextualized condition are also expected to perform worse on transfer measures.

Latency Students in the abstraction condition are expected to take less time on post-test assessments, as well as less time on tutoring problems as a result of the feedback.

Attempt Productiveness Each subsequent submission to the system (check against the compiler or test cases) will be graded against the scoring rubric for the problem. It is expected that students will have a higher average productiveness defined as the difference in scores between consecutive attempts.

Pedagogical IDE Construction

The primary technical contribution of this thesis is in the development of the Pedagogical IDE and feedback mechanism contained within it. The PIDE used in this study will be fine tuned with the results of Study 1 and 2. The students code will be evaluated against the model refined in Study 1, and tested using the techniques refined in Study 2. The system will be evaluated for accuracy against a representative sample of student code in comparison to evaluations by expert human raters.

5 Related Work

Since the beginning of computer science, there have been computer scientists building tools to help programmers be more efficient and students learn the skill of programming. Although there are still debates among hard core programmers about whether emacs or vi is the better editing environment, many professionals make use of tools that scaffold the code writing process and contain system output in nice boxes. In this section I relate previous work regarding the instructional benefits of algorithmic abstraction, research regarding contextual scaffolding in computer science classrooms and technological environments past and present to highlight the unique contributions of the Pedagogical IDE presented in this proposal.

5.1 Abstraction and Expertise in CS Education

The use of feedback regarding the algorithmic strategies employed by the students is not a new concept. Most of the early tutoring systems use this type of feedback for students within the tutoring environments[8, 10, 5, 27]. There is a large body of correlational work on the alignment between a novice's ability to make complex relational connections between the aspects of their programs[30]. The approach taken in this thesis is different from prior tools in that the feedback students will receive will be given before compilation, and an effort will be made to construct feedback that is scored as relational in the Solo Taxonomy[24] or the Block Model[20].

5.2 Contextualization in CS Education

The use of context as a scaffold for student problem solving in computer science is not a new one. Mayer[31] suggested the use of physical objects such as filing cabinets could be used to help students reason about the data structures they were trying to use in their programs. Students in Mayer's studies were working in a BASIC-like language and asked to complete tasks where either they had to write or comprehend code. The model-using group outperformed the control on tasks involving code production of looping programs, but did worse in code comprehension of looping programs. Further explorations revealed that having the students self-explain the models made them much more beneficial. In these studies students were working with languages whose syntactical requirements was fairly low. The heightened demand on students' working memory by modern languages may alter the benefit from an additional concrete model in the problem.

Current approaches to evaluating contextual scaffolding are concentrated on the effect on student motivation[12], or student performance in an entire course[15]. Much of the research involves finding the "right" context, and so contexts are compared against each other rather than the presence or absence of context.

5.3 Technological Support for the Learning of Programming

Computer science education has a rich history of successful pedagogical tools and tutors that have been shown to help improve student learning. Early tutoring systems include the Lisp Tutor[8], as well as systems for Pascal[10, 27, 32]. These systems showed that a

careful analysis of the programming process combined with other learning supports can have significant impacts on students' ability to learn programming. While these tools are effective, unfortunately they are out-moded. The programs that teach procedural or object oriented languages in their introductory courses tend to use Java or C++[6].

Current approaches to the scaffolding of programming involve mainly one of two approaches. Some systems attempt to scaffold student's problem solving through a step-by-step interaction where students are prompted for the strategies required to solve the problem[33, 9, 34]. These and other systems do show that students are able to complete the programs faster, and score higher on the assignments. These systems, however, did not show significant improvement in post-test scores. In a discussion of ProPl[9] the authors indicated, "One possible explanation is that the students were not directly responsible for the writing or organization of the design notes, in either condition. Therefore, no students had any direct practice producing decompositions written in natural language. This suggests students should be more involved with creating the design notes"[9]. Although the proposed PIDE would not require students to write design notes as suggested by Lane, the students would need to decompose the problem and attempt to implement strategies in order to solve the problem, giving them the practice in retrieving those strategies from their own mental search space.

The second approach to scaffolding programming tasks is to provide the student with a concrete representation of the code execution. There are many different types of representations used, however the most prevalent are visualizations[35]. In addition to visualizations, robots[13], media[36], and graphical environments[37, 38]. These approaches, while effective in helping students to debug and troubleshoot their programs, are only able to be used after the student achieves compilable code. Compilation is a major problem for novices[7] and as illustrated in Section 1 many semantic errors are first presented to the student as syntax problems.

There are also a number of verification strategies available for teachers and students in order to check for code correctness. The use of JUnit testing as a method for code evaluation is recognized as a useful pedagogical tool for introductory classes where Java is the language of choice[39]. This has lead to the rise of pedagogical tools including an online system called CodingBat[7][40] which is very popular among teachers. The system allows students to write short programs in a web-based environment that communicates both compile messages, and once the program compiles, success messages for a series of unit tests. Although very successful, the use of unit testing also relies on code compilation.

6 Contributions

This thesis offers the contribution of an experiment designed to infer causality of feedback using algorithmic abstraction and contextual scaffolding on student performance. The CS Education literature contains a large number of examples where students' focus on global goals of

[6]A small selection of programs use a functional style language such as scheme in their introductory courses.

[7]Previously named JavaBat, the system was renamed with the addition of Python exercises.

a program is strongly correlated with increased performance in a variety of measures[20, 41]. However, no research has been identified where instruction or feedback in global goals is experimentally manipulated to measure student performance. In domains outside of computer science, abstract questioning and context has been shown to be detrimental to students[17], and sometimes is difficult to understand by novices[18].

Prior work implies that there will be a relationship between the skill of the student and their ability to make use of the questions and feedback about the global goals of the program. Strong students will likely be able to integrate the feedback in a complex way, while weaker students may not understand either the questions or feedback and so will benefit little from the instruction. Stronger students will also benefit more from the contextual scaffolding, while weaker students may struggle with seeing the underlying deep structure of the problem through the context.

The technical contributions of this thesis offer not only an implementation of a pedagogical IDE, but a paradigm shift in the approach to providing students feedback during code production. Previous tutors and PIDEs have either treated students' code as a black box, testing with specific cases, or relied on complex models built by experts in several domains. The proposed approach in this thesis has the potential to not only impact student learning, but also the ability for educators to engage in content creation for these complex tools through rubric design.

Overall, this work has the potential to inform more than just computer science pedagogy, but also any educational domain where a computational formulation that is different from natural language is used. The influx of computational devices to the classroom is ahead of the research about how best to use those devices. More research is needed to explore what types of explicit instruction on the devices and their computational languages should be used for students.

7 Timeline

Each of the studies above seeks to address a different aspect of the research questions.

Study 1:Pilot Testing the PIDE The infrastructure for the system will be constructed during the months of June and July and tested on introductory students during July. As the model is expected to be refined over time early versions of this study may take a "wizard of oz" approach where the feedback is produced manually based on the model instead of automatically through the system.

Study 2:Evaulation of the PIDE This study is expected to begin with the system development and continue through early fall. As the think alouds will depend upon the availability of subjects it is anticipated that a full data set for evaluation will not be available until September.

Study 3:Learning Gains This study is expected to be conducted in November with students enrolled in appropriate courses.

With the above research schedule I anticipate using the fall and early spring to conduct and analyze the research and write the thesis paper. I anticipate graduating in the spring.

Bibliography

[1] Cameron Wilson, Leigh Ann Sudol, Chris Stephenson, and Mark Stehlik, *Running on Empty: The failure to teach K-12 computer science in the digital age*, Association for Computing Machinery, 2010.

[2] WestEd, Ed., *Technology and Engineering Literacy Framework for the 2014 National Assessment of Education Progress*, National Assessment Governing Board, 2011.

[3] Russell Shackelford, Andrew McGettrick, Robert Sloan, Heikki Topi, Gordon Davies, Reza Kamali, James Cross, John Impagliazzo, Richard LeBlanc, and Barry Lunt, "Computing curricula 2005: The overview report", *SIGCSE Bull.*, vol. 38, pp. 456–457, March 2006.

[4] Andrew Nusca, "Darpa: 'significant decline' in u.s. science, tech degrees 'harming national security"', *SmartPlanet*, January 2010.

[5] Elliot Soloway, "Learning to program = learning to construct mechanisms and explanations", *Communications of the ACM*, vol. 29, pp. 850–858, 1986.

[6] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers, "Program comprehension as fact finding", in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, NY, USA, 2007, ESEC-FSE '07, pp. 361–370, ACM.

[7] Matt Jadud, "A first look at novice compilation behavior using bluej", *Computer Science Education*, vol. 15, no. 1, 2005.

[8] J. Anderson and B. Reiser, "The lisp tutor: it approaches the effectiveness of a human tutor", *Lecture*, vol. 174, 1985.

[9] H. Chad Lane, Kurt Vanlehn, and H. Chad Lane, "Teaching the tacit knowledge of programming to novices with natural language tutoring", *Computer Science Education*, vol. 15, pp. 183–201, 2005.

[10] Elliot Soloway, Beverly Woolf, Eric Rubin, and Paul Barth, "Meno-ii: an intelligent tutoring system for novice programmers", *Proceedings of IJCAI'81, Proceedings of the 7th international joint conference on artificial intelligence*, 1981.

[11] Dean Sanders, "Software tools to support an objects firts curriculum", *Consortium for Computing Sciences in Colleges*, 2005.

[12] William McWhorter and Brian O'Connor, "Do lego mindstorms motivate students in cs1?", *Proceedings of the 40th ACM Technical Symposium on Computer Science*

Education, vol. 41, pp. 438–442, 2009.

[13] Stefanie Markham and K. King, "Using personal robots in cs1: Experiences, outcomes, and attitudinal influences", *Proceedings of the fifteenth annual conference on Innovation and Technology in Computer Science Education*, pp. 204–208, 2010.

[14] Eric Fernandes and Amruth Kumar, "A tutor on subprogram implementation", *The Journal of Computing Sciences in Colleges*, vol. 20, pp. 36–46, 2005.

[15] Mark Guzdial, "A media computation course for non-majors", *SIGCSE Bulletin*, vol. 35, pp. 104–108, June 2003.

[16] Jeannette M. Wing, "Computational thinking", *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[17] Kenneth Koedinger, Martha Alibali, and Mitchell Nathan, "Trade-offs between grounded and abstract representations: Evidence from algebra problem solving", *Cognitive Science*, vol. 32, pp. 366–397, 2008.

[18] Cindy Hmelo-Silver and Merave Pfeffer, "Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions", *Cognitive Science*, vol. 28, pp. 127–138, 2004.

[19] Lorin W. Anderson, David Krathwohl, Peter Airasian, Kathleen Cruikshank, Richard Mayer, Paul Pintrich, James Raths, and Merlin Wittrock, Eds., *A Taxonomy for Learning, Teaching, and Assessing A Revision of Bloom's Taxonomy of Educational Objectives*, Addison Wesley Longman, Inc., 2001.

[20] Carsten Schulte, "Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching", *Proceedings of the Fourth Annual Workshop on Computer Science Education*, 2008.

[21] J.B. Biggs and K.F. Collis, *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*, Academic Press, 1982.

[22] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad, "An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies", in *Proceedings of the 8th Austalian conference on Computing education - Volume 52*, Darlinghurst, Australia, Australia, 2006, ACE '06, pp. 243–252, Australian Computer Society, Inc.

[23] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas, "A multi-national study of reading and tracing skills in novice programmers", *SIGCSE Bull.*, vol. 36, pp. 119–150, June 2004.

[24] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students", *SIGCSE Bull.*, vol. 33, pp. 125–180, December 2001.

[25] Andrew J. Ko and Brad A. Myers, "Human factors affecting dependability in end-user

programming", *Proceedings of the first workshop on End-User Software Engineering (WEUSE)*, vol. 30, pp. 1–4, 2005.

[26] N. Charness, "Memory for chess positions: Resistance to interference", *Jounral of Experimental Psychology: Human Learning and Memory*, vol. 2, pp. 641–653, 1976.

[27] James Spohrer, Elliot Soloway, and Edgar Pope, "A goal/plan analysis of buggy pascal programs", *Human Computer Interaction*, vol. 1, pp. 463–207, 1985.

[28] Allison Tew and Mark Guzdial, "Devoping a validated assessment of fundamental cs1 concepts", *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 2010.

[29] Alexandra Johnson and Leigh Ann Sudol, "Grading our code that grades code", Meeting of the Minds: Undergraduate Research at Carnegie Mellon University, May 2011.

[30] Anne Venables, Grace Tan, and Raymond Lister, "A closer look at tracing, explaining and code writing skills in the novice programmer", in *ICER '09: Proceedings of the fifth international workshop on Computing education research workshop*, New York, NY, USA, 2009, pp. 117–128, ACM.

[31] Richard E. Mayer, "The psychology of how novices learn computer programming", *ACM Comput. Surv.*, vol. 13, no. 1, pp. 121–141, 1981.

[32] W. Lewis Johnson and Elliot Soloway, "Proust: Knowledge-based program understanding", *IEEE Transactions on Software*, 1985.

[33] E. Odekirk-Hash and J. Zachary, "Automated feedback on programs means students need less help from teachers", *Proceedings of the thirty-second SIGCSE Technical Symposium on Computer Science Education*, 2001.

[34] Amruth Kumar, "Learning programming by solving problems", *Proceedigns of IFIP Working Group, Working Conference on Informatics Curricula*, pp. 152–164, 2002.

[35] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards, "Algorithm visualization: The state of the field", *Trans. Comput. Educ.*, vol. 10, pp. 9:1–9:22, August 2010.

[36] Mark Guzdial and Barbara Ericson, "Listening to linked lists: Using multimedia to learn data structures", in *Workshop offered at the Special Interest Group on Computer Science Education*, 2011.

[37] Dean Sanders and Brian Dorn, "Jeroo: a tool for introducing object-oriented programming", *SIGCSE Bull.*, vol. 35, pp. 201–204, January 2003.

[38] Byron Weber Becker, "Teaching cs1 with karel the robot in java", *SIGCSE Bulletin*, vol. 33, pp. 50–54, February 2001.

[39] Tom Briggs and C. Dudley Girard, "Tools and techniques for test-driven learning in cs1", *Journal of Computing in Small Colleges*, vol. 22, pp. 37–43, January 2007.

[40] Nick Parlante, "Codingbat:code practice", Website, 2011.

[41] Raymond Lister, "The neglected middle novice programmer: Reading and writing without abstracting", *In Proceedings of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications*, 2007.