Exploring Variation in Students' Correct Traces of Linear Recursion

Colleen M. Lewis
Harvey Mudd College
301 Platt Blvd, Claremont, CA 91711
lewis@cs.hmc.edu

ABSTRACT

There has been a wealth of education research focused on recursion. This research has documented students' persistent difficulties with recursion, a variety of pedagogical approaches, and students' correct and incorrect mental models of recursion. This paper explores the variation in students' successful attempts to trace linear recursion. The findings go beyond correct and incorrect mental models to show how each of four modes of tracing linear recursion may require or facilitate a particular understanding of recursion. Additionally, the current study shows how knowledge of algebraic substitution can be applied to tracing linear recursion, and identifies a potential difficulty in students transferring this knowledge.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education.*

Keywords

Recursion; mental models; transfer; representations.

1. INTRODUCTION

Rather than testing a hypothesis with quantitative methods, this study used a grounded approach [1] to identify examples in which students appear to use out-of-domain knowledge to reason about computer programs [8]. The data considered came from clinical interviews [2] with thirty college students who were enrolled in an introductory computer science (CS) course. During the clinical interviews, students did a think aloud while solving problems.

The connection between algebraic substitution and tracing recursive functions became a central focus when the participant Emily (all names pseudonyms) explicitly connected a technique she used to trace a recursive function with algebraic substitution.

Inspired by Emily's statements, I looked to see if I could identify times when other participants may have been building upon knowledge from algebraic substitution. I do not assume these students consciously used this out-of-domain knowledge.

This article describes connections between algebraic substitution and four effective techniques for tracing recursive functions. To contribute a clear model of each substitution technique, I provide a description of each substitution technique using a factorial function. To demonstrate the empirical validity of the research, I provide excerpts from clinical interviews in which a participant appeared to use each substitution technique. The data and analysis presented make the following contributions:

Theoretical Contributions:

- While no assessment of prior knowledge correlates with success learning to program [10], the current study shows how knowledge of algebraic substitution can be transferred to CS, and identifies a potential difficulty in students transferring this knowledge.
- While correct and incorrect mental models of recursion have been identified [6][12], this research shows a more granular subdivision of correct mental models of recursion. The techniques presented in this article may be used as an initial taxonomy for researchers investigating students' mental models of linear recursion.

Pedagogical Contributions:

- These substitution techniques highlight an opportunity to build students' understanding of the difficult topic of recursion upon their knowledge of algebraic substitution.

2. PREVIOUS RESEARCH

This research contributes to CS pedagogical content knowledge (PCK), which is knowledge about how to teach CS and is distinct from CS content knowledge or domain-general pedagogical knowledge [4]. CS PCK research is sparse [4] and I focus on a subset of PCK seeking to understand how educators can build upon students' out-of-domain knowledge. For example, Seymour Papert recommends that students "play turtle" when programming the movement of a turtle in the programming language Logo [9]. He claims that this technique encourages students to use their "well-established knowledge of 'body-geometry'" (p. 58, [9], *i.e.* out-of-domain knowledge).

Students' understanding of recursion has been extensively studied (*e.g.,* [3][4][6][7][12]) and two mental models of recursion have been identified [6]: the copies model, a correct model of recursion, and the looping model, which incorrectly assumes that recursion is equivalent to looping. While previous research has presented a more granular taxonomy for students' incorrect attempts to trace recursive functions [12], little attention has been paid to variations within students' effective techniques for tracing recursion.

This paper builds upon the work of Leron and Zazkis [7] who described different orderings in which execution of recursive functions could be considered and described how recursion could build upon students' understanding of induction.

ICER '14, August 11 - 13 2014, Glasgow, United Kingdom
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2755-8/14/08…$15.00.
http://dx.doi.org/10.1145/2632320.2632355

Leron and Zazkis [7] generalized that mathematicians and computer scientists discuss recursive process as progressing in different directions. They claimed that "mathematicians think of the first part of the definition as a 'start rule', whereas computer scientists refer to it as a 'stop rule'." (p. 25, [7]) They provided a "likely" description of the factorial function from the perspective of both a mathematician and computer scientist. They claimed that a mathematician would justify that the "definition enables us to compute 1!, then 2!, then 3! And so on to any desired n" (p. 25, [7]) whereas a computer scientist would justify that "we can compute n! as soon as we know (n-1)!, which in turn can be computed if we know (n-2)! , and so on until we reach 1!" (p. 26, [7]).

Leron and Zazkis [7] noted the similarity between recursion and mathematical induction, which is another connection between mathematics and CS. However, induction may be no less difficult for students than recursion. This is in stark contrast to the pedagogical recommendation of this paper to build upon students' competence with algebraic substitution, which I expect is an unproblematic technique for many students.

3. METHODS

3.1 Theoretical Framework

The following four goals, which fall within the Knowledge in Pieces theoretical framework [15], encapsulate the theoretical and practical commitments that have guided the research. Each method of data collection, data analysis, and presentation of results have been shaped by these goals as referenced within sections 3.2, 3.3, and 3.4.

- **Goal #1:** Focus on individuals' understandings rather than testing hypotheses or determining the reliability of patterns.
- **Goal #2:** Focus on pedagogically relevant insights and students' correct reasoning rather than their wrong answers.
- **Goal #3:** Focus on representations created by students.
- **Goal #4:** Focus on connection between CS and out-of-domain knowledge.

Goal #1 was informed by Grounded Theory [1] and emulates methods used extensively within physics [15] and mathematics education research. Goal #2 is based upon the assumption that what some students can do, or do with help [14], is relevant PCK [4], because it is relevant for designing instruction and instructional goals. Goal #3 was inspired by research within physics [15] and mathematics [11] education about the ways representations scaffold and document students' understandings. Goal #4 was inspired by the hypothesis that college-level CS instruction could be improved if we built explicitly on students' pre-college out-of-domain knowledge [8].

3.2 Study Design & Data Collection

Aspects of the study design and data collection are listed below. Each was informed by one of the goals described above.

- **Participant Recruitment:**
  - Collecting data from a small set of students (N=30) rather than a large sample with which to test a hypothesis or reliably demonstrate a pattern (Goal #1).
  - Interviewing students during their first CS course (Goal #4).
  - Selecting interview participants who were academically successful and therefore likely to have a wealth of out-of-domain knowledge (Goal #4).
- **Data Collection:**
  - Conducting videotaped, one-on-one interviews to be able to watch and re-watch students solving problems (Goal #1).
  - Capturing video of students' hands and inscriptions rather than their faces (Goal #3).
  - Saving and scanning inscriptions students made (Goal #3).
- **Interview content and interaction:**
  - Encouraging interview participants to talk aloud to capture more of students' thought processes than whether they answered correctly or incorrectly (Goal #1).
  - Asking students a guiding question and encouraging them to retry the problem if they answered incorrectly (Goal #2).
  - Using interview questions that were found to be highly correlated with success on the 1988 AP CS exam [10] (Goal #2).
  - Asking interview participants about how things they learned outside of a CS class were helpful to them in their CS class (analysis not presented here) (Goal #4).

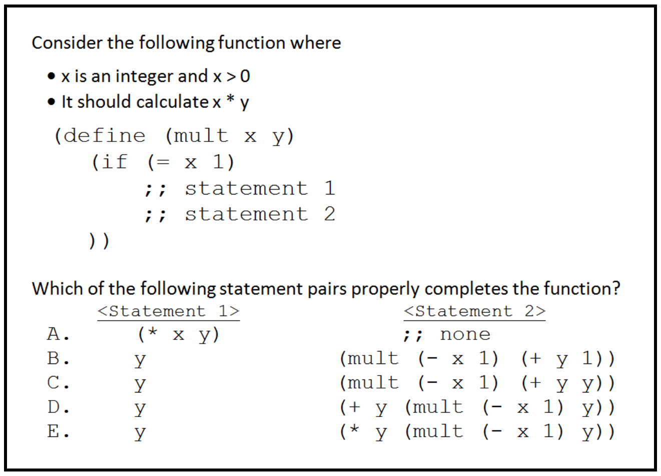
3.2.1 Interview Problems

The interview questions were questions from the 1988 Advanced Placement Computer Science A (AP CS A) exam that were highly correlated with success [10]. This article narrates participants' solutions from two of the interview problems. These problems involve the `WhatIsIt` and `Mult` recursive functions that are shown in Figure 1 and Figure 2 respectively. These questions are shown in the programming language Scheme, which was used by the majority of the interview participants.

The `WhatIsIt` function calculates the exponent x^n by multiplying `x` by itself `n` times. Therefore the correct answer for what `WhatIsIt(4, 4)` returns is answer option E, `256` or 4^4. The correct answer for the `mult` question is answer option D: Statement 1, which executes if `x` is equal to one, should be y because if `x` is one, `x` times `y` is simply `y`. Statement 2 must add `y` to the result of multiplying `y` by one less than x, which corresponds to answer option D. Notably, answer option E produces code equivalent to the `WhatIstIt` code and the two methods, `WhatIstIt` and `mult`, differ only in the names of the function and variables and the operation `*` in `WhatIsIt` versus `+` in `mult`.

```
What value is returned by WhatIsIt(4, 4)?
 (define (WhatIsIt x n)
       (if (= n 1)
              x
              (* x (WhatIsIt x (- n 1))))
a) 8    b) 16    c) 24    d) 64    e) 256
```

Figure 1. The WhatIsIt Question, a replication of a question from the 1988 APCS exam, translated to Scheme.


Consider the following function where

- x is an integer and x > 0
- It should calculate x * y

```
(define (mult x y)
   (if (= x 1)
       ;; statement 1
       ;; statement 2
   ))
```

Which of the following statement pairs properly completes the function?

```
     <Statement 1>            <Statement 2>
A.     (* x y)                 ;; none
B.     y                   (mult (- x 1) (+ y 1))
C.     y                   (mult (- x 1) (+ y y))
D.     y                   (+ y (mult (- x 1) y))
E.     y                   (* y (mult (- x 1) y))
```

Figure 2. The mult question, a replication of a question from the 1988 APCS exam, translated to Scheme.

In my explanations of the substitution techniques I will use the factorial function shown in Figure 3, which calculates the factorial of an input x in the programming language Scheme.

```
(define (fact x)
      (if (<= x 1)
             1
             (* x (fact (- x 1)))))
```

Figure 3. Example factorial function written in Scheme.

3.3 Data Analysis

The data analysis was grounded [1] and required the following activities:

- Watching, transcribing, and re-watching video to ground the analysis in data (Goal #1).
- Looking for variation between individuals and for patterns across individuals (Goal #1).
- Identifying patterns and subtlety in students' reasoning rather than coding or counting data (Goal #1).
- Analyzing students' correct inferences and answers rather than their incorrect inferences and answers (Goal #2).
- Decomposing students' correct answers to explore the variation in ways students may successfully reason (Goal #2).
- Organizing and analyzing scanned copies of students' inscriptions by question to highlight variation and themes within the representations students used (Goal #3).
- Pursuing one students' claim that she used algebraic substitution when tracing recursion to see how other students might be tacitly building upon their competence with algebraic substitution (Goal #4).

3.4 Presentation of Results

The selection and presentation of data is guided by the same four goals described in Section 3.1.

- Providing excerpts of students' solutions to show variation (Goal #1).
- Presenting hypotheses about students' ways of understanding, rather than presenting claims about generalizability (Goal #1).
- Presenting idealized substitution technique before students' data to make the techniques as clear as possible (Goal #2).
- Presenting students' correct answers, but not assuming that students have a perfect understanding (Goal #2).
- Presenting representations students made (Goal #3).
- Highlighting how algebraic substitution may be used to support each technique (Goal #4).

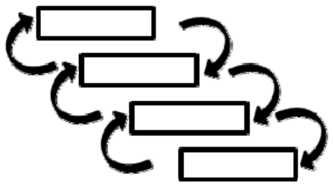
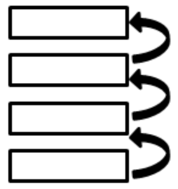
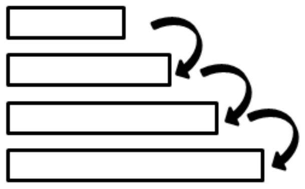
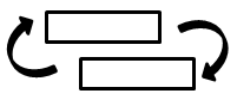
4. OVERVIEW OF TECHNIQUES

Table 1 shows diagrams of the four substitution techniques. To aid in my description of these substitution techniques I use "recursive call" to refer to a function call that is within the body of a function that calls the same function. The initial function call, which does not originate from within that function, will not be referred to as a recursive call; instead, I will refer to that as the "initial function call." I will refer to both initial functions and recursive calls as "function calls."

The top rectangle in each diagram in Table 1 represents the initial function call that is being traced by the person using this technique. The other rectangles represent recursive calls and, in some of the diagrams, include calculations that are generated during execution of that recursive call. The arrows indicate the order in which the technique requires the individual to reason about each recursive call. Each of these techniques and accompanying diagrams will be described in detail, but even without these details it is possible to observe differences between the techniques in the relative order of execution.

- *simulating execution* begins at the initial function call and then progresses down to the base case and back up.
- *accumulating pending calculations* progresses only from the initial function call down to the base case.
- *dynamic programming* starts from the base case to build up to the initial function call.
- *predicting the result* includes only the initial function call and the first recursive call, but does not consider other recursive calls or the base case.

Table 1. Table of all substitution techniques

#1: Simulating Execution 	#3: Dynamic Programming 
#2: Accumulating Pending Calculations 	#4: Predicting the Result 

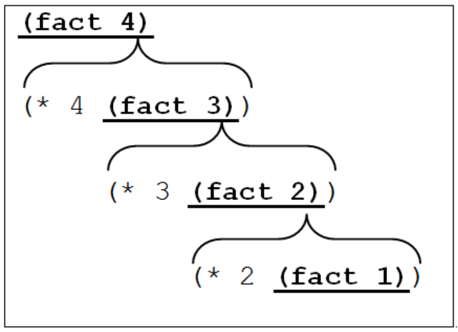
5. SUBSTITUTION TECHNIQUES

5.1 Technique #1: Simulating Execution

5.1.1 *Description*

I refer to the first substitution technique as *simulating execution*. This is the traditional method of tracing recursive functions whereby the recursive calls are traced in the order they would be executed by a computer. The output from each recursive call can be conceived of as being substituted into the expression that generated that recursive call.

For example, using the substitution technique of *simulating execution* to trace the function `fact` with the argument 4 would generate the recursive calls shown in Figure 4.



```
(fact 4)
(* 4 (fact 3))
     (* 3 (fact 2))
          (* 2 (fact 1))
```

Figure 4. Recursive calls generated by a call to (fact 4).

The underlined calls in Figure 4 are expanded from the top to the bottom. When the base case is reached, the value of 1 is substituted for the call `(fact 1)`. This is multiplied by 2 and the resulting value of 2 is substituted for the call `(fact 2)`. This is multiplied by 3 and the resulting value of 6 is substituted for the call `(fact 3)`. This is multiplied by 4 and the resulting value of 24 is substituted for the call `(fact 4)`.

Figure 5 shows my schematization of this technique. Each rectangle represents a function call. The rectangle shown on the top is the initial function call. The arrows to the right of these rectangles represent the instantiation of a recursive call. These arrows show the flow of control in a recursive function, which pauses execution within a particular function call when a recursive call is made. In a final recursive call, corresponding to the base case, in which no additional recursive calls are made, the value returned by this recursive call is provided to the calling function that had paused execution. The substitution of this return value at each step is shown with the arrows on the left of the rectangles. This also represents a change in what code is being executed. Therefore the flow of control begins at the initial function call and then proceeds to each subsequent recursive call before eventually returning from each recursive call in sequence. Each arrow is essentially an instance of substitution; the downward arrows are substitutions that work as an expansion of a particular recursive call, and the upward arrows are substitutions of return values from a recursive call. This representation is the most accurate in simulating the flow of control in a recursive function. Arrows on the right correspond to the generation of a new stack frame and each time a value is substituted, which corresponds to the arrows on the left, it returns the flow of control to the stack frame for that previous call.

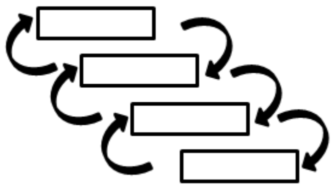
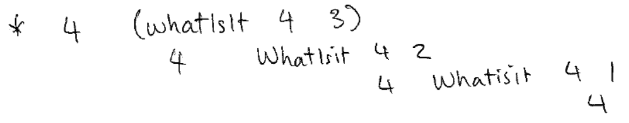


Figure 5. Diagram of the *simulating execution* substitution technique

5.1.2 *Example*

Figure 6 shows a representation created by Kate when tracing the call (`WhatIsIt 4 4`). This example was selected from the apparent instances of simulating execution because it showed the most legible and most easily interpreted representation. Each line in her representation shows the expression that would be generated by the recursive call on the previous line. However, she did not show the initial function call `(WhatIsIt 4 4)`. When the return value for each line is identified, starting from the bottom, this value can be substituted in the previous line. The participant did not identify each substitution, but summarized "and then you multiply all the fours," which is consistent with the implied substitution in the representation.



```
* 4 (whatIsIt 4 3)
      4  WhatIsit 4 2
              4  Whatisit 4 1
                          4
```

Figure 6. Written work on the WhatIsIt question by Kate

5.1.3 *Implications*

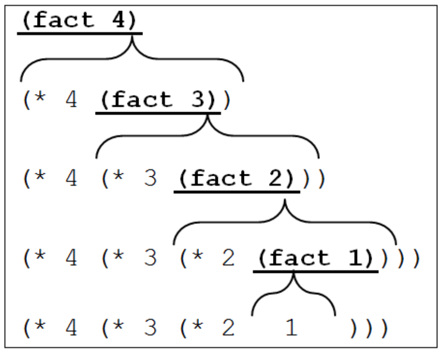
The substitution technique of *simulating execution* requires a relatively complete model of how a computer executes recursive functions. Connecting this technique to the next substitution technique of *accumulating pending calculations* may help students reason about the fact that the flow of control returns to the previous recursive calls. This feature of how a computer executes recursive calls is important for reasoning about non-linear recursion and recursion in an imperative programming environment, which both require returning to the previous recursive call to execute any remaining commands.

5.2 Technique #2: Accumulating Pending Calculations

5.2.1 *Description*

This technique contrasts with *simulating execution* in that the calculations that are performed when returning control to a paused recursive function call are accumulated in a single expression that contains all pending calculations.

For example, using the substitution technique of *accumulating pending calculations* to trace the function `fact` with the argument `4` would generate the recursive calls and calculations shown in Figure 7.



```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2   1   )))
```

Figure 7. Recursive calls generated by a call to (fact 4).

Again the underlined calls to fact in Figure 7 are expanded from the top to the bottom. However, each line includes all pending calculations. For example, the expanded version of `(fact 3)` is substituted in to the expression `(* 4 (fact 3))` to produce `(* 4 (*3 (fact 2)))`, which is shown on the third line in Figure 7. The same process generates the fourth line. Between the fourth and fifth lines the value returned by the call `(fact 1)` is substituted into the expression to produce the final expression `(* 4 (* 3 (* 2 1)))`. With this substitution technique consideration never returns to previous lines because the final expression contains all necessary calculations.

Figure 8 shows my schematization of this technique, which unlike the diagram of *simulating execution* in Figure 5 does not include an arrow indicating the flow of control returning to the calling recursive function. Each rectangle still includes a function call, but each rectangle also includes all pending calculations. In the subsequent line, the recursive call from the previous line is replaced with the equivalent, expanded, recursive relationship. Instead of representing the flow of control, each downward arrow signifies a substitution in which the recursive call is expanded and substituted in to the expression.

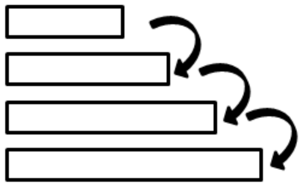
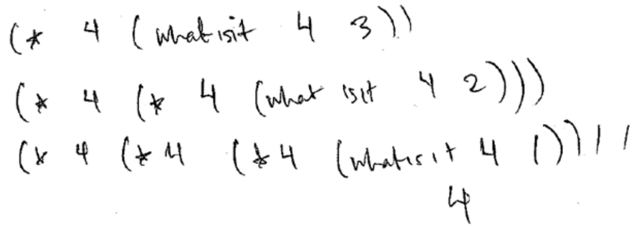


Figure 8. Schematization of the *Accumulating Pending Calculations* substitution technique

The *accumulating pending calculations* technique does not involve retracing through the previous recursive calls like the substitution technique of *simulating execution* because all pending calculations are accumulated in the final expression.

5.2.2 Example

In the next case, Jim copied the full expression each time that he substituted in an expanded expression generated by a recursive call. For example, between the first and second line he appeared to substitute "`(* 4 (WhatIsIt 4 2))`" in for the expression "`(WhatIsIt 4 3)`." He was not explicit about this process of substitution, but he arrived at the correct answer and I infer from the representation in Figure 9 that he substituted in the expanded expression from each recursive call.



(* 4 (whatisit 4 3))
(* 4 (* 4 (what isit 4 2)))
(* 4 (*4 (*4 (whatisit 4 1))))
4

Figure 9. Written work on the WhatIsIt question by Jim

Like the *simulating execution* representation shown in Figure 6, he did not write the initial call of "`(WhatIsIt 4 4)`." In the final line in Figure 9, he wrote the number 4, which he said `(WhatIsIt 4 1)` will "return." Writing only 4 on the last line instead of the full set of pending calculations, `(* 4 (* 4 (* 4 4 ) ) )` is also a departure from this technique. Despite these subtle departures, this was the most legible and most easily interpreted use of this technique.

5.2.3 Implications

The substitution technique of *accumulating pending calculations* does not require reasoning about the execution of multiple recursive calls at a time, but requires considering each recursive call in an uninterrupted sequence. This uninterrupted sequence of recursive calls is identical to the sequence of recursive calls executed by a computer. Transitioning from the use of *dynamic programming* to the use of *accumulating pending calculations* could potentially focus students on this sequence of recursive calls.

5.3 Technique #3: Dynamic Programming

5.3.1 Description

The third abstraction technique relies on calculating and storing the values of particular functions calls that build up to the initial function call. This is similar to dynamic programming, which is a technique to avoid redundant function calls.

Although the technique relies on previously calculated values, this substitution technique can be used to calculate the value for an arbitrary function call. To do this you begin by calculating the value of the recursive function for an input that does not require any recursive calls. In the case of the `fact` function, shown in Figure 3, this would be evaluating the `fact` function with an `x` value of 1. A call to `fact` with an `x` value of 1 results in evaluating the true case of the "if" statement and returns the value 1. Now we know that `(fact 1)` returns 1. Next, you evaluate the function call of `fact` with an `x` value of 2 or `(fact 2)`. This calculation results in multiplying the `x` value, 2, by `(fact 1)`. We know that `(fact 1)` returns 1 and can substitute in that value for `(fact 1)`. It would not be an example of the substitution technique if an individual instead traced the function again for the `x` value of 1. If we now know that `(fact 2)` returns 2, this resulting value from `(fact 2)` can be used when evaluating the `fact` function for the value 3. This pattern can be continued to identify the result of an arbitrary recursive call.

This process can be seen as starting at the base case and working toward the desired recursive call. Figure 10 shows a schematization of this substitution technique. For consistency with my diagrams of the other substitution techniques, I have shown the base case at the bottom of this diagram, but this diagram is not representative of the diagrams I would expect individuals to generate. In this substitution technique, the individuals' consideration of the function begins with the base case. If this was written at the top of the individuals' representation, it would generate a diagram that is an inverted version of the one shown in Figure 10.

In Figure 10 the bottom rectangle is a statement of the output of the function at the base case. For an instance of the `mult` function this would be "`(mult 1 5) = 5`." All other rectangles include an expansion of the recursive relationship, such as "`(mult 2 5) = (+ 5 (mult 1 5))`." The arrows show the process of substituting in a previously calculated value such as "`(mult 1 5) = 5`" into the expression above. After this substitution, the full contents of the rectangle would be "`(mult 2 5) = (+ 5 (mult 1 5)) = (+ 5 5) = 10`." Again, the arrows do not show the flow of control, but they show the steps of substitution of previously calculated values such as "`(mult 1 5) = 5`" or "`(mult 2 5) = 10`" into a recursive call that is farther from the base case.

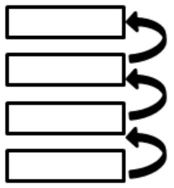


Figure 10. Schematization of the *dynamic programming* substitution technique

5.3.2 Example

Emily, who was quoted as connecting her technique to algebraic substitution, used the substitution technique of *dynamic programming* on the `WhatIsIt` problem. I will narrate a single step in her use of the strategy.

Emily was reasoning about the expression she had written that is shown in Figure 11. She had already calculated the result of `(WhatIsIt 4 1)` to be 4. In the following transcript, Emily was able to articulate how you could use the result from `(WhatIsIt 4 1)` when calculating `(WhatIsIt 4 2)`. *"I'm thinking that because we found that it was a 4 here, that it would be 4 times 4. And that would be 16."* She then paused and said *"But I think I'm over simplifying things."* I asked her to clarify and she said:

> *"Um because like when it was 4 and 1, like okay, so that was straight forward, but for when it was 4 and 2, what I was doing was like okay, If you have, when you start here. It becomes 4 and 2 minus 1, so then it's 1. So then uh oh well so we know what that is, and that was [4], so then you take it times 4."*

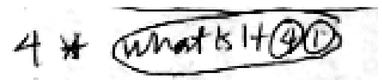

4 * whatIsIt(4)(1)

Figure 11. Previously generated tracing of (WhatIsIt 4 2)

A key element in Emily's explanation of this process is her statement *"we know what this is."* This is the central idea in the substitution technique of *dynamic programming*. Her statement *"that was 4"* stands in place of where she otherwise would have needed to explicitly trace the value of `(WhatIsIt 4 1)`. Emily proceeded to use the same technique to determine the return value of `(WhatIsIt 4 4)`.

5.3.3 Implications

The substitution technique of *dynamic programming* may be the most accessible to a novice student because the student only needs to consider a single execution of the recursive function at a time. Even without reasoning about an uninterrupted sequence of recursive calls, the student still has the opportunity to reason about the base case as producing a known value and the recursive expression producing a value that depends upon another execution of the recursive function.

5.4 Technique #4: Predicting the Result

5.4.1 Description

In the substitution technique of *predicting the result* the individual predicts the output of the first recursive call made in the body of the initial call to the function. I define "predicting the output of the first recursive call" as using a method that is independent of the recursive function to predict the output of the function. I refer to this substitution technique as *predicting the result* because the output of the first recursive call is determined "by hand" and not by using the recursive function.

If we were executing the correct version of the function `mult` with the arguments 5 and 3, the first recursive call made would be `(mult 4 3)`. This substitution technique involves predicting the output of `(mult 4 3)`. The function `mult` is supposed to multiply its arguments. Therefore *predicting the result* is trivial and `(mult 4 3)` is expected to output the value `12`, `4x3`. This expected output can be substituted into the expression in place of the first recursive call.

This requires that the individual is able to predict the output of the function and therefore requires that the specification of the function is well understood. The `WhatIsIt` function, for which the behavior of the function is not provided, is not a candidate for the use of this technique. In the example that follows, the student reasoned about the `mult` function where the expected output for various inputs could easily be identified.

This technique does not require tracing each recursive call. The initial function call is traced; however, the next recursive call is replaced with a value calculated by hand. This single step of execution is shown in Figure 12, where each rectangle represents a recursive call. The right arrow shows the flow of control that causes the first expansion of the recursive call, but the flow of control to subsequent recursive calls is not shown or considered by the individual using this technique. In the second rectangle, the value that is calculated by hand is substituted into the recursive expression. The left arrow shows the resulting value from this expression returned as the output of the function.

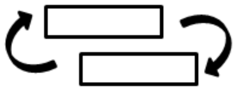


Figure 12. Schematization of the *predicting the result* substitution technique

This is a normative technique to evaluate the correctness of a recursive call, which is parallel to checking one case of an inductive chain. However, this technique does not guarantee that the recursive function is correct; the base case also needs to be correct and the recursive relationship needs to be consistent throughout the execution of the recursive function.

5.4.2 Example

In the following examples, the participant Tim used this technique twice when reasoning about the recursive function `mult` as specified by two answer options. Tim had already used the substitution technique of *simulating execution* to trace the function, but because of a systematic error in his tracing of answer option C, he could not distinguish the behavior of the functions specified by answer options C and D. The recursive relationships for these answer options are shown in Equation 1 and Equation 2 respectively. Answer option D is correct and answer option C is incorrect, but he believed them to both be correct. In the transcript below, Tim attempted a new technique, which I classify as *predicting the result* to identify whether answer option C or D was correct.

```
(mult x y) = (+ y (mult (- x 1) y))
```

Equation 1. Correct recurrence relationship specified by answer option D.

Tim created the representation shown in Figure 13 and did so without tracing individual recursive calls. There are a number of aspects of his representation that are not explicit. He created the representation shown in Figure 13 during the following interview transcript where he was tracing the correct answer, option D.

> *"Well if we look at it this way. This one's going to be y plus (wrote 'y+'), and assuming this works (pointing to answer option D) it's going to be x minus 1 times y. So it'll be like 4 y (wrote '4 y') so that'll be 5 y (wrote '= 5 y' on the second line), if we start with 5 y (wrote '5 y' on the first line). So like that should definitely work."*

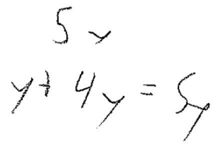


Figure 13 Inscriptions created by Tim to trace through answer option D

This technique can be used to evaluate the correctness of the recursive call as I described, but it is uncertain whether or not Tim used this technique. I interpret Tim's statements and inscriptions as indicating that he used the technique that I refer to as *predicting the result*, but continue my narration of this case by discussing some of the assumptions in my interpretation of his solution.

Tim used the inscription of "`5 y`" in both the first and second lines of Figure 13 and I interpret the meaning of them differently. In the first line I interpret "`5 y`" as representing the initial function call to `mult` with the arguments `5` and `y`, which is typically written as `(mult 5 y)`. The "`5 y`" from the second line I interpret as representing the desired output of the function call `(mult 5 y)`. It is ambiguous if his inscription of "`4 y`" should be interpreted as mathematical notation for `4` times `y` or as shorthand for the recursive call `(mult 4 y)`. However, regardless of the interpretation of this inscription his statement "assuming this works" is consistent with the use of this technique and he did not show any indication of tracing a recursive call or referring to a previously calculated value.

At this point he became *"pretty confident"* that answer option D was correct. He said: *"So I'm thinking it is more likely to be this one, and I'm just not thinking this one (answer option C) through. I think it's D. I think it's D. I'm pretty confident."* Despite his confidence, he was still unable to use *simulating execution* to show that answer option C does not also produce the correct result. After two additional attempts to trace answer option C, I encouraged him to try to see if answer option C was correct using the technique he used when creating Figure 13. The recurrence relationship from answer option C is shown in Equation 2. Using this method he convinced himself that answer option C is incorrect in the following interview transcript.

```
(mult x y) = (mult (- x 1) (+ y y))
```

Equation 2. Incorrect recurrence relationship specified by answer option C.

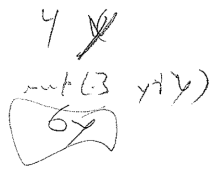


Figure 14 Inscriptions created by Tim to trace through answer option C

> *"Alright, well that reasoning is – that in theory multiply works. And does what we want. So if we start with four and um. Four y (wrote '4y'). When you run this, it's going to give us multiply (wrote 'mult') three times y plus y (wrote '(3 y + y)'). And three times y plus y. Three times two y is six y (wrote '6 y' and drew a box around it). So that's not right. And that would convince me I'm wrong (referring to his conclusion that answer option C was correct)."*

Like the first example where he proceeded by "assuming this works," here he explained his reasoning as "that in theory multiply works and does what you want." Using answer option C, Tim again traced a single execution of the recursive call specified by answer option C and substituted in the expected output of the `mult` function. Again he appeared to represent the initial function call of `(mult 4 y)` as "`4 y`." He was explicit about the recursive call to `mult` that would result. Applying the recurrence relationship from answer option C shown in Equation 2 and wrote "`mult (3 y+y)`."

5.4.3 Implications

The substitution technique of *predicting the result* builds upon students' experience reasoning about algorithms, which may be another connection to students' out-of-domain knowledge. A barrier to reasoning using the other techniques is the layering of multiple steps. This technique requires considering only two function calls and therefore may be less overwhelming to a student.

6. DISCUSSION

The techniques identified in this article are not intended to be comprehensive of all possible techniques for reasoning about recursive functions. As a counter example, one technique, instead of using substitution, involves seeing that the algorithm is the same as the algorithm known by the individual to perform the same calculation. This mapping allows an individual to conclude that the recursive function works as expected, but does not involve tracing specific values. This technique was used by only a single participant on the `mult` problem, despite participants' likely familiarity with the algorithm for multiplication.

When reasoning about the `mult` problem, Peter describes "`x*y`" as *"really saying x plus x plus x, y times."* He observed that answer option D "looks like it might do that" and came to the correct conclusion that answer option D was the correct answer without ever tracing the function. During the segment documented in the following interview transcript, Peter created the representation shown in Figure 15.

*"Umm, I can't really explain it, but this seems reasonable as an answer. (Interviewer: OK, Why?) Umm, because I kind of think of multiplication as if we have x times y (wrote 'x*y' shown in Figure 15) that's really saying x plus x plus x, y times (completed inscriptions in Figure 15). So, and this looks like. Looks like it might do that, but I'm not sure (pause). So statement 1, umm. I guess that would imply this (points to answer option D), so I guess I'll try D out first."*

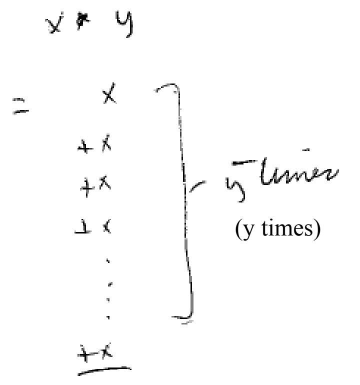


Figure 15. Peter's notes when explaining why answer option D was correct

After making the conclusion that answer option D "seems reasonable as an answer" Peter used *simulating execution* to trace through each of the answer options with sample input. Although Peter did not use the technique to determine his final answer, this is a valid technique with which to reason about the function and does not involve substitution.

7. CONCLUSIONS

This article includes descriptions and empirical examples of four kinds of substitution techniques that I refer to as *simulating execution*, *accumulating pending calculations, dynamic programming*, and *predicting the result*.

The diversity of these techniques may be a tool to make CS more accessible. Students may find various techniques more or less intuitive and may be able to use a single technique as an intuitive foothold to build toward a rich understanding of recursion.

The substitution techniques of *simulating execution*, *accumulating pending calculations*, and *dynamic programming* all trace each recursive call, but do so in different orders. I hypothesize that the differences between these three techniques may provide the opportunity to scaffold students' understanding of how recursive functions are executed by computers and may provide the opportunity to highlight relevant features of the execution of recursive functions. For example, these substitution techniques and the corresponding representations for tracking execution could be taught to students, which may support students in more accurately tracing execution.

8. ACKNOWLEDGEMENTS

This work was partially funded by NSF grants #1044106 and #1339404. The paper also benefitted from feedback provided by anonymous reviewers, Michael Clancy, Andrea diSessa, Katherine Lewis, and Arielle Schlesinger.

9. REFERENCES

[1] Corbin, J. M., & Strauss, A. C. (2008). Basics of Qualitative Research. Thousand Oaks, CA: SAGE Publications.

[2] diSessa, A. A. (2007). An interactional analysis of clinical interviewing. *Cognition and Instruction.* 25*(4)*, 523-565.

[3] George, C. E. (2000) Experiences with Novices: The Importance of Graphical Representation in Supporting Mental Models. In A. F. Blackwell & E. Bilotta (Eds). Proc. PPIG 12

[4] Hubwieser, Magenheim, Muhling, & Ruf (2013). Towards a Conceptualization of Pedagogical Content Knowledge for Computer Science. *ICER.* 1-8.

[5] Kahney, H. (1989) What do novice programmers know about recursion? *Studying the Novice Programmer* (E. Soloway & J.C. Spohrer, Eds.) Hillsdale, New Jersey: Lawrence Erlbaum Associates Inc., 209-228.

[6] Kurland, D. M., & Pea, R. D., (1989). Children's mental models of recursive logo programs. *Studying the Novice Programmer (E. Soloway & J.C. Spohrer, Eds.)* Hillsdale, New Jersey: Lawrence Erlbaum Associates Inc., 315-323.

[7] Leron, U. & Zazkis, R. (1986). Computational Recursion and Mathematical Induction. *For the Learning of Mathematics, 6(2)*. 25–28.

[8] Lewis, C. M. (2012). *Applications of Out-of-Domain Knowledge in Students' Reasoning about Computer Program State.* (Doctoral dissertation). Retrieved from ProQuest Dissertations and Theses. (Accession Order No. 12710).

[9] Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic Books, Inc.

[10] Reges, S. (2008) They mystery of b := (b = false). *ACM SIGCSE*, 39, 21-25.

[11] Schoenfeld, A. H. (2007). Reflections on an assessment interview: What a close look at student understanding can reveal. In A. H. Schoenfeld (Eds.) *Assessing Mathematical Proficiency* (pp. 267-280). Cambridge: Cambridge University Press.

[12] Scholtz & Sanders (2010). Mental Models of Recursion: Investigating Students' Understanding of Recursion. *ITiCSE*, 103-107.

[13] Simon *et al.* (2006). The ability to articulate strategy as a predictor of programming skill. *Proc Eighth Australasian Computing Education Conference,* Hobart, Australia.

[14] Vygotsky, L.S. (1978). Mind in society: The development of higher psychological processes.

[15] Wagner, J. F. (2006). Transfer in Pieces. Cognition and Instruction, 24(1), 1-71.