Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers

Kathi Fisler
Brown University
Providence, RI
kfisler@cs.brown.edu

Francisco Enrique Vicente Castro
WPI
Worcester, MA
fgcastro@cs.wpi.edu

ABSTRACT

When functional programming is used in studies of the Rainfall problem in CS1, most students seem to perform fairly well. A handful of students, however, still struggle, though with different surface-level errors than those reported for students programming imperatively. Prior research suggests that novice programmers tackle problems by refining a high-level program schema that they have seen for a similar problem. Functional-programming students, however, have often seen multiple schemas that would apply to Rainfall. How do novices navigate these choices? This paper presents results from a talk-aloud study in which novice functional programmers worked on Rainfall. We describe the criteria that drove students to select, and sometimes switch, their high-level program schema, as well as points where students realized that their chosen schema was not working. Our main contribution lies in our observations of how novice programmers approach a multi-task planning problem in the face of multiple viable schemas.

KEYWORDS

Rainfall; program schemas; functional programming

1 INTRODUCTION

Soloway's Rainfall problem [17] has become a benchmark in computing education. This problem (which essentially asks students to compute the average of a sequence of numbers that appear before a sentinel value) is interesting because it appears straightforward while having non-trivial underlying complexity. Over the years, several authors have noted some of the challenges with Rainfall (see section 2), leading some to question whether the community is making progress on "beating" the Rainfall problem [8].

Most existing work on the challenges of Rainfall was conducted in the context of imperative programming [15–17, 20]. Some researchers have begun to publish studies of Rainfall with students who used functional programming [7], but those studies have not reported particular challenges that arise when students attempt Rainfall in this context. Given that different programming languages have different idioms and affordances, a better understanding of how students solve—and struggle with—Rainfall in different pedagogic contexts and programming languages will enhance our understanding of this deceptively interesting programming problem.

The functional perspective is particularly interesting because students who learn functional programming are typically exposed to *multiple viable solution structures* for Rainfall. Studying how students approach Rainfall with functional programming thus provides an opportunity to explore how novice students navigate multiple applicable schemas, each of which they may only partly understand from CS1. Formally, the research question explored in this paper is:

> *When novice programmers have seen multiple schemas that might apply to a problem, how does their solution emerge and evolve?*

We explore this question qualitatively, through narratives of four students' attempts at Rainfall in a talk-aloud session at the end of a CS1 course. These studies exposed factors in how novice students select, switch, and apply program schemas to problems requiring plan composition.

2 RELATED WORK

Most published studies of Rainfall involved students who were programming imperatively, in languages such as Pascal [17], Java [15], Python (also [15]), or C++ [16]. Ebrahimi had groups of students working in various languages, one of them Lisp [5]: however, his students had learned imperative constructs within Lisp.

Fisler published the first study of Rainfall with functional programming [7]. Her data was from multiple schools that were using the *How to Design Programs* (henceforth HTDP) curriculum, but with a variety of languages (Racket, OCaml, and Pyret). In her sample of 218 students, 186 had a (mostly) correct solution to Rainfall. Furthermore, her participants produced at least three different high-level structures of solutions (two appear in section 5). In contrast, imperative studies have typically produced a common high-level structure, taking only a single pass over the input sequence (whether with `for` or `while` loops), maintaining the running sum and item count in variables [15–17] (also seen in our own imperative studies). The study participants in this paper were also learning HTDP, but they are from a school that did not participate in Fisler's original study.

Castro and Fisler [1] captured the computer screens of HTDP students working on a different plan-composition problem called *Adding Machine*. That problem asks for a list of sums of sublists of input as separated by zeros, stopping when two consecutive zeros are discovered. Castro and Fisler's students performed quite poorly on the problem, with many of them following the HTDP design processes into an initial program structure that was not suitable to solve the problem. One participant in this study shared

ICER '17, August 18-20, 2017, Tacoma, WA, USA

ACM ISBN 978-1-4503-4968-0/17/08…$15.00
https://doi.org/10.1145/3105726.3106183

this problem; others avoided it, but could have gone down this path. Section 7 discusses our students' design processes in detail.

Pirolli et al.'s studies of learning recursive programming observed that novices modify already-learned solutions to fit the context of a new problem [11, 12]. Spohrer and Soloway's studies of the end-product programs of students and their talk-aloud protocols (verbal reports of planning, implementation, and debugging steps taken in programming a solution) echo this [19]: they suggest that students either (1) use previously learned programming knowledge (programming plans) to write the code, or (2) translate relevant non-programming knowledge (non-programming plans) into code. Students repair coding decisions after testing uncovered unexpected code behavior. Rist [13, 14] refined these models, describing two paths programmers take when writing code. When a programmer knows a viable schema, she takes a *plan retrieval* path, implementing code in *top-down* fashion, with smaller-scale modifications to address problem subtleties. When the programmer has no schema in her memory, she takes a *plan creation* path. She identifies a core computation called a *focus* (usually, a major computation required in the problem such as adding in an averaging problem) writes code to implement the focus, and then builds around that code *bottom-up* until a working solution is achieved. None of these theories of novice program construction addresses what happens when programmers have weaker knowledge of multiple viable schemas, or how novice programmers switch schemas mid-stream. This is the main question explored in our study.

de Raadt et al. used Rainfall to study impacts of explicitly teaching planning strategies [3]; they do not discuss change in strategy.

3 BACKGROUND: THE RAINFALL PROBLEM

Soloway proposed the Rainfall problem in the 1980s in the context of studying student difficulties with plan composition [17]. The original wording asked students to compute the average of a sequence of numbers, which were input through keyboard I/O, that occurred before a sentinel value had been entered. Soloway identified four sub-tasks that needed to be composed: taking in input, summing the inputs, computing the average (which involves counting the inputs), and outputting the average. Over the years, other researchers studied variations of Rainfall: some added noisy data in the form of negative numbers, some added additional reporting requirements (such as printing the maximum daily rainfall as well as the average). All variations have shared common core goals of summing, counting, averaging, input, and output.

Our formulation of Rainfall includes noisy data, but only asks for the average as output. We provide the inputs in a data structure, as our host course does not teach I/O. Our version reads:

> Design a program called rainfall that consumes a list of numbers representing daily rainfall readings. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list representing faulty readings. If you cannot compute the average for whatever reason, return -1.

This version requires six tasks:

- Sentinel: Ignore inputs after the sentinel value

```
; A list-of-number is
; - empty, or
; - (cons number list-of-number)

;; TEMPLATE for list-of-number (generic name lon-func)
#|
(define (lon-func alon)
  (cond [(empty? alon) ...]
        [(cons? alon) ... (first alon)
                      ... (lon-func (rest alon)) ...]))
|#
```

Figure 1: The HTDP input-type description and template for a program to process a list of numbers. The input-type description is a comment (semicolon is the comment character in Racket, the stick/hash create a block comment). cons creates a new list from an element and an existing list (it does not modify the existing list). The template has one conditional branch for each variant in the datatype (here, empty list and non-empty list). In the non-empty case (marked by cons?), the template recurs on the rest of the list to guarantee traversal of all elements. The ellipses get filled when the template is used to solve a specific problem.

- Negative: Ignore negative inputs
- Sum: Total the non-negative inputs
- Count: Count the non-negative inputs
- DivZero: Guard against division by zero
- Average: Average the non-negative inputs

4 BACKGROUND: THE *HOW TO DESIGN PROGRAMS* CURRICULUM

HTDP [6] is an introductory computing curriculum that teaches students how to leverage the structure of input data and multiple representations of functions to design programs. Students are taught a multi-step process called the *design recipe* for approaching a new programming problem. Roughly, the steps include identifying the type of input and output data for a problem, writing concrete examples of the input data, writing a type signature (though in comments rather than a formal type language) for a function that solves the problem, writing several illustrative examples or test cases for the function, writing a skeleton of code (called the *template*) that fully traverses the input type but ignores details of the desired output, and filling in the template with details of the given problem. Figure 1 shows the type description and template for a list of numbers, the datatype used in Rainfall.

The template is the most relevant aspect of the design recipe for this study. Templates provide schemas for programs. Unlike some schemas, which are contextualized to a style of problem, templates mirror the structure of a particular data type. In the early part of an HTDP course, students are taught to always start with the template that matches the type of their input data. Later in the course, as programming problems get more complex, students learn other schemas and the contexts in which to use them (thus relaxing, or at least broadening, the template step of the design recipe). Students in our study course had been exposed to two other schemas that could apply to Rainfall; we discuss these in section 5.

Week	Topics	Assignment
1	Arithmetic expressions and functions	Composing images
2	Conditionals and structures	Functions over structs (movie theater data), conditionals, test cases
3	Lists of atomic data, the design recipe	Functions over structs (capturing weather events), functions over lists of strings
4	Lists of structs	Lists of structs (political ads)
5	Trees	Binary search trees
6	Locals and higher-order functions	N-ary trees (system of rivers and tributaries)
7	Accumulators	map and filter (revisit political ads), accumulators (variant on numeric max)
8	Variables, mutation	None (end of course)

Figure 2: The topic sequence in the host course for this study. Our Rainfall talk-alouds occurred at the end of week 7.

In HTDP, functions and data types are the building blocks for programs, not variables and loops as in curricula based on imperative programming. Courses start with writing non-parameterized expressions to compute with numbers and images (e.g., composing images to create flags), then teach abstraction over concrete similar expressions to create functions in roughly the third lecture. Students then cover a series of data structures—structs/records, lists of atomic data, lists of structures, binary trees, n-ary trees—each following the same design recipe to scaffold program design based on the shape of the input data structures. All of this material precedes mutable variables (covered much later in the course).

The HTDP Instance for this Study

The course in which we conducted the study was a CS1 course for students with limited or no prior programming experience (those with prior experience take a different course). The course uses Racket (a variant of Scheme) as its programming language. Figure 2 outlines the sequence of topics and assignments in the course. The course ran over 8 weeks, with 4 hours of lecture per week and one hour of lab per week. The Rainfall study occurred during week 7.

The course did not explicitly cover plan composition or decomposing problems into sub-tasks. The course did emphasize creating *helper functions* to break down larger computations, with appropriate use of helpers counting significantly in homework grading. Prior to the Rainfall session, every problem covered in lecture or assigned for homework had either been a structural traversal of a recursively-defined data structure (a list, a binary tree, or an n-ary tree), or a function that used a single additional parameter to accumulate one running value (such as the sum of elements so far in a list). In particular, students had not yet seen a problem that wasn't a direct instantiation of an HTDP template.

5 RAINFALL UNDER HTDP

At first glance, Rainfall seems a natural fit with HTDP: the problem involves straightforward functions over lists of numbers (counting and summing, both canonical recursive functions that students see in lecture when lists of numbers are introduced). The fit is less clear, however, in the context of the template. If a student followed the basic recipe blindly, applying the list-of-numbers template, they would start with the following code:

```
;; rainfall : list-of-number -> number
;; compute average of non-neg nums before -999
(define (rainfall alon)
  (cond [(empty? alon) ...]
        [(cons? alon) ... (first alon)
                      ... (rainfall (rest alon)) ...]))
```

This code is hard to modify into a working Rainfall solution: because rainfall is called recursively on the rest of the list, filling in the ellipses in the `cons?` case requires computing the average of a list from the average of the rest of the list. This is more complicated than the usual algorithm of dividing the sum of the list by its length. Each of sum and count are straightforward applications of the HTDP template, but the rainfall function itself needs to decompose its computation into these two sub-tasks.

HTDP exposes students to multiple viable Rainfall solutions (which is what makes this study interesting in the first place). Figure 3 shows a solution that reduces the input data to the list of numbers to average (truncating at -999 and removing the negatives), then calls a function to average the clean list. Observe that the `sum` and `actual-rain` functions follow the list-of-numbers template, but the overall `rainfallC` function does not (it decomposes the average task into the sum and count tasks instead). This structure was the most common in Fisler's earlier Rainfall study with functional programming [7].

Figure 4 shows a solution structure based on what HTDP terms *accumulators*. This program includes a nested function with parameters for each of the running count and sum of data to average. Once the end of the data or -999 is reached, another local function is called to produce the average. This structure traverses the data only once, and is closer in style to what an imperative programmer would produce with a loop and variables for the sum and count.

6 STUDY LOGISTICS

Our data were collected as part of a course-long study of how CS1 students (at a selective US university) approach program design. In one session of this study, 13 students talked aloud as they spent 30 minutes trying to solve Rainfall (as defined in section 3). After 30 minutes, we archived the student's code and interviewed them about how they approached the problem: what they found difficult, what information they drew on, and what inspired their design decisions. Both the talk-aloud and the interview were audio-recorded, then transcribed verbatim for analysis. Students worked on a computer, in the course's standard programming environment.

Participants: This paper presents data from four students from the overall study (additional narratives did not fit in the page limits). Of the four, two are female and two are male. We selected these four to reflect variety in course performance, prior experience, and the structure of students' final solutions. Figure 5 summarizes each student's grades and programming experience prior to CS1.

```
;; sum : list-of-number -> number
;; produces the sum of the given list of numbers
(define (sum alon)
  (cond [(empty? alon) 0]
        [(cons? alon) (+ (first alon) (sum (rest alon)))]))

;; actual-rain : list-of-number -> list-of-number
;; produces list of non-negative values that occur before -999
(define (actual-rain alon)
  (cond [(empty? alon) empty]
        [(cons? alon) (cond [(= (first alon) -999) empty]
                            [(negative? (first alon)) (actual-rain (rest alon))]
                            [else (cons (first alon) (actual-rain (rest alon)))])]))

;; rainfallC : list-of-number -> number
;; produces average of non-negative nums in list before -999, or -1 if no such nums exist
 (define (rainfallC alon)
  (local [(define good-data (actual-rain alon))]
    (if (> (length good-data) 0)
        (/ (sum good-data) (length good-data))
        -1)))
```

Figure 3: Rainfall solution in Racket, clean-first style. The overall function (`rainfallC`) calls a helper function (`actual-rain`) to truncate and clear negative numbers from the input data. It then computes the sum and length to compute the average. `length` is a built-in operator that returns the length of a list. Semicolon is the Racket comment character. This solution could be adapted to use higher-order functions: `fold` can compress the sum to a single expression, and `filter` could be used to remove the negatives if a separate function had been used to truncate data after -999. [18] humorously presents a Scala version.

```
;; rainfallA : list-of-number -> number
;; produces average of non-negative nums in list before -999, or -1 if no such nums exist
(define (rainfallA alon)
  (local [;; produce-average: number number -> number
          ;; computes average given count and sum, producing -1 if count is 0
          (define (produce-average count sum)
            (if (> count 0) (/ sum count) -1))

          ;; rainfall-accum: list-of-number number number -> number
          (define (rainfall-accum data count sum)
            (cond [(empty? data) (produce-average count sum)]
                  [(cons? data)
                   (cond [(= (first data) -999) (produce-average count sum)]
                         [(negative? (first data)) (rainfall-accum (rest data) count sum)]
                         [else (rainfall-accum (rest data) (+ 1 count) (+ (first data) sum))])]))]
    (rainfall-accum alon 0 0)))
```

Figure 4: Rainfall solution in Racket, accumulator style. The overall function (`rainfallA`) contains two local (nested) function definitions: one for computing the average and one that recurs through the input list, accumulating the sum and count of non-negative values until -999 is reached. Semicolon is the Racket comment character.

ID (Gender)	Experience	Exam	Course
StudA (F)	C++, Online courses	71 (C)	76.47
StudB (M)	Java, Python, Ruby, Self-study	87 (B)	94.36
StudC (M)	Python, JavaScript, Java, HTML5, CSS, PHP, Self-study, AP class, High school class, Online courses	89 (B)	81.08
StudD (F)	None	93 (A)	87.14

Figure 5: Participant overview. Experience was self-reported via checkboxes on a survey. The exam was in course week 3.

Data Analysis: The first author developed the narratives in section 7 from the typed talk-aloud and interview transcripts. She is an experienced HtDP instructor, but did not teach this instance of the course. The second author conducted the talk-alouds and interviews. He reviewed the first author's narratives for accuracy. We divided work this way so that the narratives would reflect the pedagogy and learning of HtDP more than personalities of the students. The first author does not know the identities of the students.

The narrative methodology here is influenced in part by the narrative analysis method used by Whalley and Kasto in their investigation of novices' code writing strategies [21]. They developed descriptive accounts of how students used existing schema to write

code from think-aloud and interview data. We also draw on ideas from grounded theory [9], in terms of the narrative reconstructions that describe how students varied in how they chose constructs, patterns, or techniques to build their Rainfall solutions.

In the analysis, we marked comments pertaining to choice of schemas, choice of language constructs, discussion of design choices, mentions of problem tasks (whether or not they were reflected in code), and rationale for editing previously-written code. We also marked comments on how students perceived the Rainfall problem.

7 NARRATIVES

This section presents narratives of each participants' design process[1]. We also summarize both the correctness and the structure of each final solution. Possible correctness values are *poor* (far from working), *fair* (in the right direction, but with many errors), and *almost correct* (very close and could have been fixed easily after some straightforward testing to show the bugs). We show final code for some students, but space precludes including it for all.

7.1 StudA

Correctness: poor
Overall Structure: Accumulator, but role of parameter unclear

StudA begins by writing the function name and input type. She proceeds to write the list-of-numbers template (as in fig. 1). Inside the non-empty list case, she inserts a conditional to check whether the first element of the list is positive.

She thinks of using an accumulator in order to track the running sum of positive numbers. She goes back to her notes to check on how to write an accumulator function, then adds a local definition for a function with an accumulator parameter. She recalls that accumulator functions return the accumulator parameter in the base case; accordingly, she replaces the -1 she was originally returning in the base case with the accumulator parameter.

She notes that *"I can do the division at the end"*, then goes back to working on the running sum. If the first list element is negative, she calls the function recursively with the same parameter value. She returns to thinking about where to handle the division: *"I feel like the division should happen inside the function. So I don't want to be adding here ... I want to divide the rainfall - actually no wait I want to add the rainfall"* (at this point, she is wrestling with how to integrate the sum and average tasks within the same area of code).

StudA notices that her current code never returns -1 (by inspection, not by running it): *"So now my issue is nothing will turn up -1 if the average can't be produced or if the list is just empty. So somehow I have to work that in there."* She decides to try running the code. She tries an input of all positive numbers, but gets back a negative average. She realizes this can't be right. She correctly articulates that an average is computed by dividing the sum by the count.

After this point, StudA starts to thrash. She articulates a variety of possible edits involving -1 and the accumulator parameter. Her comments include statements like *"somehow I have to store the divided value into the accumulator or to make that produce at the end."*. She continues to try to reason out how her code works. She realizes that there are multiple subtasks: *"somehow I have to get the three of these things together without adding all three together"*. She seems to keep switching the task (addition, division, counting, or returning -1) to do around the recursive call on the rest of the list—her final code (below) reflects this confusion. Just before time is up, she thinks of using a separate helper function: *"So maybe I need to make a helper function where I just add them all up and then divide out later."*. Time runs out before she can try it in code.

```
(define (rainfall alon)
  (local
    [(define (rainfall alon acc)
      (cond [(empty? alon) acc]
            [(cons? alon)
             (if (> (first alon) 0)
                 (/ (rainfall (rest alon) (first alon))
                    (+ 1 acc))
                 (rainfall (rest alon) acc))]))]
    (rainfall alon 0)))
```

During the interview after the coding session, StudA remarks *"I thought accumulator would be useful because every time it finds another positive value [...] the average changes because the bottom number would keep getting bigger. So the accumulator would keep adjusting to that."* The student has associated some behavior with accumulators, but does not understand the pattern well enough to get close to a working solution.

7.2 StudB

Correctness: fair (count of data inaccurate)
Overall Structure: Accumulator with parameter for running sum

StudB begins by writing the function name, input type, and output type as he reads the problem. The student starts to follow the template by creating a conditional, articulating that the function should return 0 if the list is empty (this appears to be a pattern of habit, as the correct answer on empty input would have been -1). As the student is thinking out what to do when the list is non-empty, he articulates the algorithm for computing the average, and says *"we want to divide something by the length of [the input list]"*, observing that only the non-negative values should be considered.

The student realizes he needs a helper function that sums the values in a list. The student articulates the type signature and writes a sum function following the HTDP template for lists of numbers. This function does not account for negative numbers or the -999 sentinel. The student then goes back to the original function and starts to handle the negatives, introducing a conditional that checks the sign and value of the first number on the list. When -999 is encountered, the student notes that the program should return the average (but doesn't completely fill in the needed code). As the student continues filling in the conditional, he starts to question whether the helper could be handled by built-in primitives.

The student finishes filling in the conditional and tries running the program, but discovers it goes into an infinite loop. At this point, the buggy program follows the pattern to recursively sum the positive numbers, returning the average when the -999 is encountered: one branch of the conditional within the recursion is implementing the sum task while another implements the average task (which can't work since this leaves no base case for the sum task). The student realizes that the code isn't *"storing the value"* of

[1] We follow Dziallas and Fincher [4] in calling these narratives, not case studies.

the running sum, and switches to an accumulator-based design, with a parameter to hold the running sum.

The student then begins a cycle of editing the code, testing it, having the tests fail, then editing again. As the student talks through the cycle, he begins looking for fragments of code to delete or modify: for example, he tries removing various branches of conditionals, including the one that terminates the recursion if the list becomes empty before reaching -999 (this branch never gets restored before time is called).

Next the student tries to figure out where to return -1: *"So this is still working but this is not working. So it's not producing -1. And so if the element's negative it's running the recursion on the rest of the list. Maybe - no. Maybe the [accumulator] could be set to something else other than just [the current accumulator value] or but I can't think of what it needs to be set to."* The student hits on the idea of a different helper function to handle the case in which all numbers in the original input list are negative. He proceeds to write a straight-up (correct) recursive function to check whether all numbers in a list are negative, then uses this to guard computation of the average once -999 is detected. That said, the student never got the tasks and their code mapping straight in his head. He kept modifying the in-progress code with Rist-like focals, rather than thinking about how to decompose the problem.

The final code contains two major errors: it does not handle input lists that lack the -999, and the average computation uses the wrong denominator (the length of the suffix that follows the -999, not the count of non-negative numbers before the -999).

7.3 StudC

Correctness: fair (conflates sum and average tasks)
Overall Structure: Accumulator with filter (latter not integrated)

StudC starts by writing the function signature and purpose. He begins to write the list template, filling in -1 as the answer in the empty-list case based on the problem statement. He wonders whether he should be using `local`, which is part of the standard pattern for writing functions with accumulators in the course. The student starts to write the inner accumulator function, again following the template. But this time, the student returns the accumulator value in the empty-list/base case. That is the standard usage pattern students have seen with accumulator functions to this point in the course. To this point, StudC has not articulated what the accumulator variable represents; his work seems entirely syntactic.

The student talks about checking whether the first number in the list is negative, then about creating a helper function to compute the average; this comes up more as a side comment than as part of the flow of where this helper might get called from the overall Rainfall computation. The student realizes that the average computation will need both the running sum and the count of items, and thinks about how to obtain both values: *"it almost seems like I would use an accumulator to show how many times I've actually gotten through that. [...] so I guess we'll use another local"* (whether the student is suggesting another locally-defined accumulator function or another parameter within the existing accumulator is not clear at this point).

The student notes the requirement to stop at the first -999 and to ignore negatives. The student recognizes that `filter` could ignore the negative numbers, and would eliminate the need to check the sign of individual list elements during the accumulator function. The student writes a helper function that uses `filter` to remove all non-positive numbers from an input list. (The student does not, however, call this helper function from the accumulator function. The helper remains uncalled in the final code).

Next, the student adds a conditional to check for a value *less than* -999 (incorrect logic, changed in final). For the "then" branch, the student articulates calling the function recursively to process the rest of the list, while adding the new value to the accumulator. As shown in the final code below, the student adds another parameter (`times`) to track the count of values. He tries to compute the average and use it as a new parameter value (he never articulates a clear role for this parameter). The else case of the conditional gets a recursive call to the function that takes the rest of the list and leaves the two accumulator parameters unchanged.

```
(define (rainfall alon acc times)
  (cond [(empty? alon) acc]
        [(cons? alon)
         (if (> (first alon) -999)
             (rainfall (rest alon)
                       (/ (+ (first alon) acc) times)
                       (+ 1 times))
             (rainfall (rest alon) acc times))]))
```

The student then enters a testing phase, running his code on a single test case. The test fails. The student correctly diagnoses that the execution never satisfies the -999 check and reverses the less-than computation in his conditional check. The student adjusts initial values for his accumulator parameters, but does not correctly trace the execution to isolate the actual errors in his code.

7.4 StudD

Correctness: Almost correct (sans two `cond` cases reversed)
Overall Structure: Clean-first with accumulator (for cleaning)

StudD begins by writing the function name, input type, and output type as she reads the problem statement. She proceeds to start writing the template, inserting -1 as the answer in the base case based on the problem statement. She instinctively questions whether the base case answer should instead be 0, but decides to follow the problem statement and see where it goes. She does not appear to write the non-empty case of the template blindly, but instead talks through what might need to happen in this case.

She fairly quickly ponders whether she will need an accumulator, but she isn't entirely sure why this would be necessary. She thinks she should have a function that *"goes through each number in the list just to make sure it's not -999"*. She goes on to say that *"with every number that it passes that is not -999, it's gonna add those all up"*. So at this point, StudD has decided to write a function that traverses the list and adds up all the relevant data.

StudD begins to change course once she thinks about what to do upon finding the -999: *"so then I would need another helper function. Once it hit the -999, it would divide it by the [...] number of terms it went through but I don't know how I would do that yet"*. As she tries to write the base case of her accumulator function, she realizes that summing and the overall rainfall problem require different base-case answers: *"if it's empty, that would return either–it would return -1 for the rainfall purposes, but for this one I don't know if it would return 0 [or] -1"*. This prompts her to change her accumulator

to instead build a list of the relevant (clean) data, with separate functions to compute the average of this list. Her final solution is a clean-first style, but with an accumulator in the function that cleans the data. During the reflection interview, she remarks how accomplished she feels for solving the problem.

8 ANALYSIS AND DISCUSSION

This paper opened with a specific research question: *when novice programmers have seen multiple schemas that might apply to a problem, how does their solution emerge and evolve?* All four students started saying they would use the list template and ended up using accumulators in some fashion. Whether the students perceived these as different patterns, or whether they view accumulators as a variation on the list template, is not evident in our transcripts. However, all four students commented on the typical base cases of these patterns, suggesting that they had internalized them separately.

The trigger to use accumulators differed across the students: StudA and StudB initially associated the accumulator with tracking the sum (though StudA lost this association once she started to thrash); StudC switched without a clear justification and never stated a purpose for the accumulator (following the schema purely syntactically). StudD explicitly ruled out an accumulator at first, then found it useful for tracking clean data. Use of accumulators was likely influenced by the timing of our Rainfall session. The course had just covered accumulators: the pattern was fresh and students may have assumed they *should* be using them. The lectures had shown the use of accumulators for summing a list of numbers.

As discussed in section 2, we have not found existing theories about how novices navigate or switch between multiple schemas. Observations from our data suggest possible elements of such theories, each raising open questions that would inform a theory.

Observation 1. *Students who copy-and-paste the template (as* htdp *recommends for beginners) get more stuck than those who recall the template and write it down "as they go".*

StudA mechanically wrote down the list-of-number template before thinking about the details of Rainfall. The course teaches this practice, though once students have mastered the template, they tend to interleave writing the template with filling in the holes (particular in easy spots, such as the base case). StudB, StudC, and StudD all stated that they were going to use the list-of-number template, but they proceeded to work in "write as you go" fashion, which meant they started thinking about how they would fill in the holes around the recursive call to Rainfall before they committed to calling their function on the rest of the list. These students generally introduced an accumulator at this point, effectively switching their program schema mid-session. StudA, in contrast, struggled more with the schema change and ended up farthest away from a working Rainfall solution.

Open Question 1. *When students know multiple schemas for a problem, do those who write out most of one (incorrect) schema have a harder time adapting their approach than those who reproduce schemas on the fly?*

Open Question 2. *Is there a systematic method for helping students realize when they might need to switch schemas? Or how to recognize apriori when a problem needs more than the basic schemas that they know?*

Observation 2. *Students who articulated only the syntactic schema of accumulators, but not the underlying concept, struggled to adapt them to the needs of Rainfall.*

As instructors, it is easy to assume that once students have seen the *idea* of a parameter that accumulates a running value, then they will add as many such parameters as a problem requires. This assumes that students understood the underlying idea, however, rather than simply absorbing the syntactic pattern. Students in our course had only seen examples with a single accumulator parameter, and in each of those programs, the value in that parameter was returned in the base case of the recursion. An accumulator-based Rainfall solution either needs two parameters (one for the running sum and one for the running count) or one parameter for the running list of clean data. Students had only seen examples that accumulated numbers up to this point in the course. Unless students understood the point of the accumulator, adapting to multiple parameters could be a significant challenge.

Interpreting this in an imperative context, it would be as if students had only ever seen programs with a single numeric variable, and did not immediately realize that they could have two variables. This is not a confusion that we have seen reported in other Rainfall studies. In functional programming, additional "variables" become additional parameters—perhaps that seems more complex to novices than additional standalone variables (which could be ignored while still allowing the program to run, whereas additional parameters need values or a syntax error results). Perhaps students in the imperative studies of Rainfall made different errors depending on whether they had seen programs with multiple variables. The point here is simply that different linguistic constructs have different affordances and pitfalls, and different courses prepare students for problems in subtle ways that we have likely overlooked in reporting studies. We need to understand our benchmark problems in multiple contexts to know what makes them challenging.

> StudA*: I guess [the hardest part] was trying to figure out how to work in the -1 with the accumulator there because I didn't know where to put it [...] all the examples we put the accumulator after empty [...] but in this one the answer wasn't stored in the accumulator.*

Observation 3. *Students who connected accumulator parameters or parts of their code to specific tasks, and maintained those connections through the schema switch, produced more correct code.*

The two students with clear roles for the accumulator were also the ones who more generally connected specific problem tasks to parts of their code. One of these was the only student who mentioned using `filter` to help deal with the negative numbers (though he never got that part integrated with his accumulator-based program for computing the average). These observations reinforce the idea that failure to decompose problems into tasks—not just failure to compose code—underlies student challenges with multi-task problems (others' work showing that students can handle similar problems when explicitly taught strategies or patterns supports this [3, 10]). Had someone suggested decomposing the problem into separate sum and count functions, we suspect the two weakest

students might well have done better, since their transcripts showed they did have basic facility with the list template.

Observation 4. *Students had not understood that each sub-task that traverses a list needs its own function or accumulator parameter.*

Both HtDP and the host course explain that a single recursive function can perform only one traversal-based operation (this initially comes up when discussing insertion sort, to explain why separate functions are needed for insertion and the overall sort). Our host course did not, however, reinforce this via assignments. Accumulator parameters enable a single function to track outputs of multiple tasks in a single traversal, but the course does not currently teach the explicit link between traversal-tasks and parameters. Our narratives show students struggling to integrate multiple traversal tasks (e.g., summing and counting) in a single function, even once they introduce accumulators. The connections between tasks, parameters, templates, and traversals are not (or have not been made) clear enough to these students, yet they seem critical to producing a correct Rainfall solution in any programming language.

Students similarly struggled to handle the sentinel. All prior problems in the course terminated a list recursion at the end of the list, not at a particular value. Most students recognized the sentinel as another base case for recursion, but they struggled to reconcile the return values in the empty-list and sentinel cases, especially in light of the -1. This is again a failure to separate tasks in their code.

Open Question 3. *Would more emphasis on the "one task per function or parameter" rule enable students to solve Rainfall, even if they hadn't seen sentinels or multiple accumulator parameters?*

The first author has begun teaching students how to identify tasks and map them to each of functions or parameters/variables as part of program design. We are in the early stages of studying the impact on students' abilities to solve plan-composition problems.

Observation 5. *Students thought the problem was complex just from the problem statement.*

Our version of Rainfall has more constraints and detail than Soloway's original phrasing, which read:

> *Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.*

Later versions of the problem have included negative numbers, but even compared to those, our problem description has additional details such as: (a) -999 may never appear, (b) -999 may appear more than once, (c) an explicit instruction to return -1 if the average cannot be computed, and (d) use of the term "faulty readings" to contextualize the other negative numbers.

Prior versions of the problem typically omitted instructions on what to return if there is no data to average (detail (c)). We agree with Seppälä et al. [15] that this omission makes it hard to interpret students' mistakes. While throwing an error would be better than returning -1, our students had not yet learned error handling.

Details (a) and (b) regarding the sentinel are necessary because the input comes as a list rather than being entered interactively. Requiring the list to contain -999 actually complicates the problem for one who follows HtDP (or any datatype-based discipline) strictly. A list with a guaranteed sentinel would have a different data type (in which the base case is a list with the sentinel as the first element, not the empty list); this would lead to a different template. The current wording retains the schema that students already know. Taken together, however, all of these details have a price in terms of how students perceive the problem complexity:

> StudC: *From what we've learned in class we generally use just simpler problems, and we rarely [...] put them all together. So when you are approached with a problem such as this, you almost struggle to figure out how to put it together 'cause you've never done it before. [...] [usually] it would be more in a Part A, Part B, Part C, Part D style.*

Future research should explore relationships between the level of detail in the problem statement, whether examples are provided [15], and when students perceive sub-tasks in more complex problems:

Open Question 4. *Does a more detailed Rainfall description help students recognize more sub-tasks before they start coding?*

Open Question 5. *Does providing examples or test cases with the Rainfall description lead students to recognize more sub-tasks before they start coding?*

9 CONCLUSIONS

Our study data allowed us to ask a unique question in the context of Rainfall: how do novice students manage having seen multiple viable schemas for a programming problem? Students do not yet know the limitations of these schemas well (unlike experts). We would expect, then, to see students switching schemas or perhaps trying to merge them. We are not aware of theories of how novices switch schemas. We need to understand this, however, so we can teach students how to handle such situations more effectively.

Our data drive home the power—and hold—of previously-seen patterns for novice programmers. Instructors may think they are teaching a general approach (such as using an accumulator), but if students have only seen examples that use that approach in a single way (such as a single parameter that is returned as the final answer), they may struggle to adapt patterns to new situations. Approaches such as subgoal labeling [2] might help counter the syntactic power of a pattern. In the context of this paper, the key takeaway is that the class examples instructors choose may inadvertently complicate problems like Rainfall for students. If we want to know what makes Rainfall hard or easy, we need to consider the course context at a finer granularity than has been reported in previous studies.

Our students did not seem to perform as well as those in Fisler's study [7]. Our participants had only just started working with higher-order functions, which many students used in Fisler's study. Perhaps curricular differences addressed our observations for Fisler's study courses. It would be interesting to run similar talk-alouds with students at the schools from the original study.

As a community, we can't claim to have "beaten the Rainfall problem" [8] until we have findings that we can explain and reproduce across courses. This needs studies that report on finer-grained curricular details and how students draw on them when selecting designs. While we continue to work on those, programming instructors should continue to carry an umbrella.

ACKNOWLEDGMENTS

Work supported by US-NSF Grant No.s 1116539 and 1500039.

REFERENCES

[1] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 205–210. DOI: http://dx.doi.org/10.1145/2839509.2844574

[2] Richard Catrambone. 1998. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General* 127 (1998), 355–376. DOI: http://dx.doi.org/10.1037/0096-3445.127.4.355

[3] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95 (ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 45–54. http://dl.acm.org/citation.cfm?id=1862712.1862723

[4] Sebastian Dziallas and Sally Fincher. 2016. Aspects of Graduateness in Computing Students' Narratives. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, New York, NY, USA, 181–190. DOI: http://dx.doi.org/10.1145/2960310.2960317

[5] Alireza Ebrahimi. 1994. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies* 41 (1994), 457–480.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs.* MIT Press. http://www.htdp.org/

[7] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*. ACM, New York, NY, USA, 35–42. DOI: http://dx.doi.org/10.1145/2632320.2632346

[8] Mark Guzdial. 2010. A Challenge to Computing Education Research: Make Measurable Progress. https://computinged.wordpress.com/2010/08/16/a-challenge-to-computing-education-research-make-measurable-progress/. (Aug. 2010). Accessed April 14, 2017.

[9] Päivi Kinnunen and Beth Simon. 2012. Phenomenography and grounded theory as research methods in computing education research field. *Computer Science Education* 22, 2 (June 2012), 199–218. DOI: http://dx.doi.org/10.1080/08993408.2012.692928

[10] O. Muller, B. Haberman, and D. Ginat. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of ITiCSE*.

[11] Peter L. Pirolli and John R. Anderson. 1985. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology/Revue canadienne de psychologie* 39, 2 (1985), 240–272.

[12] Peter L. Pirolli, John R. Anderson, and Robert G. Farrell. 1984. *Learning to program recursion.* 277–280.

[13] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989), 389–414.

[14] Robert S. Rist. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Hum.-Comput. Interact.* 6, 1 (Mar 1991), 1–46.

[15] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15)*. ACM, New York, NY, USA, 87–96. DOI: http://dx.doi.org/10.1145/2828959.2828963

[16] Simon. 2013. Soloway's Rainfall Problem Has Become Harder. *Learning and Teaching in Computing and Enginering* (2013), 130–135. DOI: http://dx.doi.org/10.1109/LaTiCE.2013.44

[17] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858.

[18] Juha Sorva and Arto Vihavainen. 2016. Break Statement Considered. *ACM Inroads* 7, 3 (Aug. 2016), 36–41. DOI: http://dx.doi.org/10.1145/2950065

[19] James C. Spohrer and Elliot Soloway. 1989. *Simulating Student Programmers.* Morgan Kaufmann Publishers Inc., 543–549.

[20] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Computing Education Research Workshop (ICER)*. 117–128.

[21] Jacqueline Whalley and Nadia Kasto. 2014. A Qualitative Think-aloud Study of Novice Programmers' Code Writing Strategies. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, NY, USA, 279–284. DOI: http://dx.doi.org/10.1145/2591708.2591762