On the Interplay Between Bottom-Up and Datatype-Driven Program Design

Francisco Enrique Vicente Castro
Department of Computer Science
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA, USA
fgcastro@wpi.edu

Kathi Fisler
Department of Computer Science
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA, USA
kfisler@wpi.edu

ABSTRACT

When students are faced with a programming problem unlike any they have solved before, prior research suggests that they develop code backwards from essential computations in the problem. Some curricula, however, teach students to first write scaffolding code based on the type of the input data. How do these two approaches interact? We gave CS1 students who were taught to write scaffolding code a programming problem unlike any they had seen before. We found that while students put essential computations into the scaffolds, they often overuse affordances of the scaffolds in ways that lead to plan-composition errors. We propose that steering students away from on-the-fly decomposition while programming could help avoid some of these errors.

Keywords: Novice programmers; models of code development; plan composition

1. INTRODUCTION

Most models of how novices program suggest that they use previously learned examples or solutions as starting points for new programs [3, 4, 10]. What happens, however, when a new problem is sufficiently different that previously-learned examples don't apply?

In studies with students learning Pascal, Rist [5, 6] determined that novices write down a statement or expression that captures the essence of some program task (the *focus*), then work backwards to integrate this into the overall program. In contrast, pedagogic approaches such as *How to Design Programs* (henceforth HTDP) aim to be more systematic, teaching students to first write scaffolding code that exploits the structure of input data. These two perspectives, one a model developed from observing students programming procedurally and one a process designed to scaffold traversal of recursively-defined datatypes (as taught in some CS1 courses that use functional languages), could either complement or interfere with one another. What role does each play in helping students develop correct solutions to new problems? What hindrances does each introduce?

SIGCSE '16, March 02 - 05, 2016, Memphis, TN, USA

ISBN 978-1-4503-3685-7/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2839509.2844574

This paper explores the interplay of bottom-up programming and datatype-driven design, using Rist's model and HTDP as concrete instances of each. We report on a study in which HTDP students were given a problem that required program structures they had not yet learned. We video-recorded their sessions, looked for how students used each of focal-expressions and HTDP scaffolds in attempting to solve the problem, and extract insights on the affordances and limitations of each approach. In part, this work explores how bottom-up programming manifests in functional programming. In part, it tries to identify ways to leverage each perspective towards developing pedagogic techniques that help students tackle new programming problems.

2. BACKGROUND

2.1 Rist's Focal Expansion Model

Several papers from the mid 1980's proposed models of how students create new programs by adapting known solutions to similar problems (we review these in section 2.3). Rist's *focal-expansion model* [5, 6] expanded on these to cover situations in which students lacked similar problems on which to build. His model identifies two states that students can enter when encountering a programming problem:

1. **Plan Retrieval**: If a novice knows a solution to a similar problem, she will retrieve it (from memory) and reproduce the code in a top-down fashion.

2. **Plan Creation**: If a novice does not know a solution to a similar problem, she will start from a small code fragment for an identified computation within the problem. This code fragment is called the *focus* or *focal computation*. She will then expand the code around the focus in bottom-up fashion to integrate the new code into the rest of the program.

When a known solution applies to a more difficult problem, the model predicts that students switch to bottom-up creation mode after retrieval. As a programmer's experience increases, she makes heavier use of retrieval.

Intuitively, the focus is the essential computation for an identifiable *task* within the program. For the task to sum a sequence of numbers, an expression like `sum := sum + n` would be the focus (where `n` would be defined subsequently, perhaps by reading the next input from the sequence). In the problem of averaging a sequence of numbers, there are three focal expressions, one for each of the required tasks of summing the elements, counting the elements, and dividing

the sum by the count. Rist's model does not address which focal a student would handle first. Rather, it claims that one of the expressions `sum := sum + n`, `count := count + 1`, or `average = sum / count` would be written and expanded upon first, rather than more generic code such as "iteratively read input" (which would only occur first in retrieval mode).

Rist developed this model from watching students produce code (in Pascal) for problems such as calculating the volume of a box-like house or sorting weights into ascending order. The essence of the model lies in (a) new plan creation starting from a focal computation, and (b) the construction of code being either top-down or bottom-up, depending on whether a plan is being created or retrieved.

2.2 How to Design Programs

How to Design Programs (HTDP) is a CS1 curriculum that teaches a step-by-step "recipe" for designing programs [2]. Its two core ideas are *write test suites first* and *design programs from the shape of the input data*. A student trained in HTDP first writes down the input and output types of a function, followed by several examples (test cases) of the function's behavior (including both inputs and outputs), followed by a *template* for the function. The template captures code that is dictated by the shape of the data, leaving holes to fill with problem-specific computations.

As an example, consider a function to determine whether a list of strings contains "pie". A student would first write the type signature (as a comment) and several examples (a.k.a. `check-expects`). We present code in Racket, as that is the language used in both our study and the HTDP textbook (though the principles apply more generally).

```
; containsPie? : list-of-string -> boolean
(check-expect (containsPie? empty) false)
(check-expect (containsPie? (list "apple" "pie")) true)
(check-expect (containsPie? (list "bread" "tea")) false)
```

Next, the student would write the *template*: code that exhausts what is known about the shape of the input data (in this case a list of strings):

```
; containsPie? : list-of-string -> boolean
(define (containsPie? alst)
  (cond [(empty? alst) ... ]
        [(cons? alst) ... (first alst)
                      ... (containsPie? (rest alst)) ]))
```

The template has a conditional that checks whether the list is empty. If it is, the template simply contains a hole for the function's result in that case. If not, then the list must have both a first element and the subsequent elements (rest). The latter is itself a list of strings, and hence should also be processed by `containsPie?`. The template therefore includes a recursive call on the rest of the list. The template has holes in place of concrete code for combining the first element with the result from the recursive call. Note that nothing in the template is specific to the computation required for `containsPie?`: the template simply traverses the input data.

To finish the function, the student fills in the holes with details specific to `containsPie?`. The student can leverage the test cases to do this (details of how to do this are not relevant to this paper). The next box shows the final code: the student filled in `false`, a comparison to the string `"pie"`, and the `or` operator to complete the function.

```
; containsPie? : list-of-string -> boolean
(define (containsPie? alst)
  (cond [(empty? alst) false]
        [(cons? alst) (or (string=? "pie" (first alst))
                          (containsPie? (rest alst)))]))
```

Templates are at the heart of the difference between HTDP and Rist's model. They scaffold development of code, giving students systematic rules to follow to get beyond a blank page when starting a programming problem (in Rist's creation state). HTDP teaches students concrete rules (not described here) for creating templates from datatypes; these rules scale to rich data structures including binary and n-ary trees, graphs, and other mutually-recursive data. This set of uniform rules across datatypes provides a detailed *process* for designing programs, particularly on new datatypes. As students write multiple functions over the same datatype, templates can also serve as *schemas* for other programs over that datatype. Many students appear to internalize these schemas, without having to use the rules to create templates afresh each time.

2.3 Other Related Work

Early studies by Pirolli *et al.* on novices learning recursive programming observed that people rely heavily on known solutions when developing new programs [3, 4]. Novices modify already-learned solutions to fit the context of the new problem. 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 [10]. They suggest that students, when writing code for a problem, 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 then proceed to a testing phase where they detect problems (or impasses) within their code, then enter a debugging phase to repair these impasses.

Spohrer and Soloway observed that most novice programming mistakes are due to difficulties with *plan composition*, the putting together of program fragments to form a working program [9]. They analyzed buggy programs in terms of their *goals* relative to their *plans* (groups of code that work together [7, 8]). Goals and plans are cognitively plausible, deep structure knowledge that programmers have, based on knowledge of working programs. For example, programmers do not perceive an expression like `sum += sum` merely as an assignment statement (code-specific: surface structure), but as part of a plan for a running sum (deep structure). Reasoning through a goal/plan perspective, and not merely through surface-structure language constructs, considers chunks of code as single units. Spohrer and Soloway identified several issues that make plan composition difficult for novices, including cognitive load problems, unexpected cases, and optimization problems, among others [9].

3. AN EXPLORATORY STUDY

By design, HTDP templates should defer entry into Rist's "creation" mode, giving students a schema to retrieve based solely on the type of input to a function. If students have not solved a similar problem to the overall function, Rist's model predicts that students would then write focal computations. However, Rist's notion that students write the focal compu-

tation then build context around it differs from the HTDP process, in which students would place focal computations either in template holes or in auxiliary (a.k.a. helper) functions. Seeing what HTDP students do after writing templates should give insights into whether and how Rist's model, and the general idea of bottom-up programming, play out in the context of data-traversal schemas for recursive programs.

To better understand the interplay between Rist's model and HTDP, we conducted a study in a single HTDP-based CS1 class at a university in the USA. We gave students a programming problem over an input datatype they knew (a list of numbers), but that required additional programming techniques that they had not yet seen. This combination should have put students into Rist's creation mode (perhaps after retrieving the template). We video-captured the students' programming sessions, then analyzed the videos to see when students wrote each of HTDP templates and Rist's focal computations, and how they edited around these pieces to complete the program.

In particular, we sought insight on the following questions:

1. When do HTDP-trained students use templates?
2. How does Rist's idea of focal computations manifest in HTDP programs?
3. How and when do HTDP students integrate focal computations into existing code?

Our questions attempt to avoid bias in favor of either Rist's model or HTDP's claimed benefits. While we expected students to follow HTDP's process (as this was a key part of the course), we did not assume that students had internalized that process enough to actually do so. This is an exploratory study, asking whether (a) Rist's focal-expansion theory applies to functional programming through HTDP and (b) HTDP provides useful scaffolding to students on problems that require significantly different programming techniques than what they have already seen.

3.1 The Problem: Adding Machine

Our study used a programming problem called *Adding Machine* that consumes and produces a list of numbers:

> Design a program called `adding-machine` that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.
>
> Example:
> `(adding-machine (list 1 2 0 7 0 5 4 1 0 0 6))`
> should produce `(list 3 7 10)`

Adding Machine involves four tasks:

a) Ignoring data after the double-zero
b) Identifying sublists separated by single zeros
c) Summing the elements in each sublist
d) Building the output list from the sums of the sublists

Students trained in functional programming would write a recursive solution. Viable recursive approaches include:

- **Reshape the data first**: The sublists in the input are embedded in a flat list of elements delimited by zeros. The input could be reshaped into a list of lists that omits the zeros. For example, the input `(list 1 2 0 7 0 5 4 1 0 0 6)` could be reshaped as `(list (list 1 2) (list 7) (list 5 4 1))`. Separate functions would iterate over this list and sum each individual sublist.

- **Accumulate sums in a parameter**: The recursive function could take an additional parameter for the sum of the current sublist. When a 0 is detected at the front of the input list, this parameter would be concatenated onto the result of processing the rest of the input list with the sum parameter initialized to 0.

- **Recur on a new list containing the sublist sum**: Since the recursive function takes the list to process as an input, the first position of the list can be used to store the running sublist sum. For example, the call `(adding-machine (list 1 2 0 5))` would generate the call `(adding-machine (list 3 0 5))`. This approach is distinctly functional, as imperative solutions rarely modify a list mid-iteration. Special care is required, however, if a sublist can sum to 0.

- **Recur on a new list that skips the first sublist**: A function could recur on the suffix of the list without the first sublist, using a separate function to produce the sum of the prefix corresponding to the first sublist.

Detecting the consecutive-zero termination pattern adds a bit of complexity, as solutions must check both the length of the remaining input (an input that doesn't contain the 00 pattern might have only one element) and the values of the first two elements. Solutions can either truncate the input data at the double-0 in a separate pre-traversal, or integrate checking for the pattern into the core computation.

This problem seems excellent for studying the interactions between Rist's model and HTDP. Lists of numbers are a familiar datatype to HTDP students: many will have already internalized the schema for flat lists. Each of the high-level solutions outlined above, however, uses some more-advanced programming concepts: parameters that accumulate data or recur on something other than the rest of the list are covered after trees in HTDP. While basic mastery of lists would suffice to reshape data, students would not be exposed to the *idea* of doing so until much later. Thus, if HTDP students are given this problem after a couple of weeks of programming with lists, they will have a schema that appears to apply (the basic list template), but no experience with "similar" solutions that draw on these more advanced concepts. Many students should end up in the plan-creation state while working on this problem, even if they initially retrieve the list-of-numbers template.

3.2 Data Collection and Logistics

We collected data in Spring 2015 in a CS1 course taught using HTDP in Racket. Neither author was on the course staff. Students were given roughly 40 minutes to work on *Adding Machine* during a weekly lab session. Each student used the SnagIt video-capture tool to record all activity within the window for the course IDE (DrRacket). Students uploaded both the video and their final source-code file at the end of the lab session. In total, 138 students submitted data; we randomly sampled 25 to analyze in this study. In terms of final course grades, the sampled population earned

```
1 Test AM 3
2 Template-list AM
3 AM buildsumlist (cons (helper1 first-L)
                        (AM rest-L))
4 Template-list helper1
5 helper1 sumelts (+ first-L (helper1 rest-L))
6 helper1 singlezero (= 0 first-L (AM rest-L))
7 AM singlezero (= 0 first-L)
8 Test helper1 3
```

```
;; ListofNumber->ListofNumber
;; adds together elements of a sublist and returns
     them as a list
(check-expect
     (adding-machine (list 1 2 0 7 0 5 4 1 0 0 6))
     (list 3 7 10))
(check-expect (adding-machine empty) empty)
(check-expect
     (adding-machine (list 5 15 22 0 7 0 8 1))
     (list 42 7 9))

(define (adding-machine lon)
  (cond [(empty? lon) empty]
        [else (cons (findzero (first lon))
                    (adding-machine (rest lon)))]))
```

Figure 1: Sample coding sequence (top) and the actual program code (bottom)

5 As, 13 Bs, 3 Cs, 3 fails, and 1 incomplete. We thus had a good mix of students relative to mastery of the material and likelihood of needing help.

We conducted the study five weeks into the academic term, after students had roughly 16 lectures (50-minutes each), 4 labs (50 minutes each), and 4 multi-exercise programming assignments for homework. Before this point, the course had covered defining and calling functions, composing functions, conditionals, recursive functions over lists, and the HTDP design process (including test-first development of functions and templates, as described in section 2.2). The students had written several functions over lists of numbers, strings, and records prior to doing the study. The course had not yet covered trees, accumulating results of computations in additional parameters, or recursive calls on an argument other than the rest of the list.

3.3 Coding Programming Edits

When analyzing the videos, we recorded several events:

1. **Template use**: Students wrote an HTDP-prescribed template for a new function.
2. **Task-specific computation**: Students wrote code related to one of the four problem tasks. We recorded the actual code, the task it belongs to, and the function in which students put the code. The tasks were recorded with the following labels:
   - **singlezero**: handling single-zero delimiters
   - **doublezero**: handling the double-zero sentinel
   - **sumelts**: summing elements (of a sublist)
   - **buildsumlist**: building the list of sublist sums
3. **Test case use**: Students wrote test cases for a specific function. We recorded the function name and number of tests written for it before the next event occurred
4. **Other**: Students made edits that did not fall into one of these categories

Figure 1 shows an example of our analysis summary, along with its corresponding Racket program (the program shows the code as of step 3 in the summary). In the summary, `AM` refers to the *Adding Machine* function. The student wrote another function named `findzero`, replaced with the alias, `helper1`, in the summary. Names for helper functions were replaced with aliases with the format `helper<number>` to facilitate consistency in the coding as the students would sometimes change the names of helpers as they programmed their solutions. We produced a summary such as this for each of the 25 sampled programs.

4. ANALYSIS AND INTERPRETATION

None of the sampled students produced working solutions for *Adding Machine*, despite evidence that they used templates, developed focals, and tried to decompose the problem. Plan composition was the main hurdle, particularly when students tried to reuse template code inappropriately in multiple smaller-scale plans.

All but one student (24 of 25) used the list-of-numbers template (same as the `containsPie?` template in section 2.2). From there, students took many approaches. Table 1 shows two examples of the directions students took: the student on the left created a helper function to handle both single-zero delimiters and computing the sum of sublists, while the student on the right tried to handle both the double-zero and sum tasks within the template for *Adding Machine*. Both show plan-composition errors which we will explain as we describe general patterns in our data.

4.1 Focals After Templates?

We hypothesized that students would enter creation mode after retrieving the template. As such, we looked at what code students wrote immediately after the template and where they put it, checking whether it captured focals. As Table 2 shows, all but 3 students wrote expressions that took on specific tasks. Most (19) students put this new code into the template for the *Adding Machine* function. This matches focal-expansion theory, as well as HTDP pedagogy. Most template holes get filled by focal-like expressions, though some decomposition (through helpers) also goes there.

Whether students had entered creation mode, however, isn't clear. Summing a list is a standard HTDP programming problem; students may have retrieved the `+` code as done in Table 1 (right). The single- and double-zero tasks are about termination of computations. As such, they resemble base cases of recursive functions (even though the usual base case of a recursive function on a list handles the empty list). Students may have retrieved the pattern of terminating a traversal, adapting it to recognize patterns of zeros. The left student in Table 1 wrote base-case-like code to catch the single-zero in the `adder` function.

Overall, 21 students who started with templates immediately filled in holes in that template with focal expressions for a specified program task and 2 began to decompose the problem by creating a helper function (in both cases to handle *Double zero*). Of the remaining 4, one wrote a focal computation (*Single zero*) within a non-template function, and 3 wrote something not clearly linked to a problem task.

4.2 Plan Composition

Whether students retrieved or created plans for the problem tasks, they still had to compose them into an overall

Table 1: Code samples: (Left) A task identified is pulled out into a separate function; (Right) Interleaving function calls within one function without decomposition.

```
(define (adding-machine lon)
  (cond [(empty? lon) empty]
        [else (cons (adder lon)
                    (adding-machine (rest lon)))]))

(define (adder lon)
  (cond [(empty? lon) 0]
        [(= 0 (first lon))
             (adding-machine (rest lon))
        [else (+ (first lon) (adder (rest lon)))]))
```

```
(define (adding-machine lon)
  (cond
    [(empty? lon) 0]
    [else
      (if (and (= (first lon) 0) (= (second lon) 0))
          (list 0)
          (+ (first lon)
             (adding-machine (rest lon))))]))
```

Table 2: First tasks coded, with location (*Note: T = within template, NT = not in template*)

Task	Within AM	New Helper
Sum sublist	6	0
Output list	3	0
Single zero	9 (T) + 1 (NT)	0
Double zero	1	2
None/Other	3	

Table 3: Where code for each task appears

Task	Just in Main	Just in Helper	Both
Single zero	7	3	10
Double zero	4	1	2
Sum sublist	9	5	6
Output list	8	0	0

program. Here, students displayed significant difficulties. Each solution in Table 1 illustrates one of these challenges.

The solution on the right tries to integrate the plans for sum and double-zero by sharing template code: the template base-case (the `empty?` check) returns a 0 as in the sum plan, while the "new" base case for double-zero returns a list (the output type of the overall function). In attempting to share the recursive call (which would be syntactically identical in both the sum and truncate-at-00 plans), the student created an inconsistency in the output type of the program. Only 5 students even wrote both the sum and double-zero tasks; 3 of these put these tasks in the same function. Six students put both the single-zero and double-zero tasks in the same function, missing that each terminates a different other task (processing a sublist and identifying input to process, respectively).

In the solution on the left, the student tried to decompose the problem via a helper function. The output task stayed in the main template, while the sum and single-zero tasks moved into the helper. This approach was on the right track, but had two key errors (aside from the missing plan for double-zero): the single-zero detection needed to be a base case (and return 0) in the helper, and the recursive call in the main function needed to take the suffix without the first sublist as input (rather than the entire rest of the list). Despite these flaws, this student at least had a largely consistent view of the output type of each function (the erroneous single-zero base case answer notwithstanding). This reflects an understanding (missed in the solution on the right) that one use of template code can return only one type of output.

Eleven students moved the sum-sublist task into a helper (as in the solution on the left). None modified the portion of the list passed on the recursive call, instead using the recursive call verbatim from the template. Overall, 16 students created a helper function that took a list of numbers as input and included some program tasks. Table 3 summarizes the number of students who covered each task in each of helper functions, the main *Adding Machine* function, or both.

Even when students realized to create helpers, they often failed to effectively decompose the problem around those helpers: 12 of 25 created helpers that they never called from their main function. These helpers attempted combinations of the single-zero, double-zero, and sum-sublist tasks. This seems a different manifestation of thinking through focals: rather than integrate a focal computation into an existing function (their original templates), students tried to put them in separate functions. This is not unreasonable, as each of these three tasks involves traversing a list, and students had been taught to use recursive functions to traverse lists. Of the students who created helper functions, 8 used templates in writing all helper functions (2 more did so sometimes), again suggesting a strong HTDP influence. These observations suggest that upon entering the creation state (after setting up templates), students resort to building their functions with a characteristic tinkering behavior [1] by patching up the holes in the template with familiar constructs and function calls, even when these result in output inconsistencies and essentially, plan composition problems.

Several students put the same tasks in both the main and helper functions (last column of Table 3). Task-replication seems to depart from Rist's focal model, which suggests that students would write the focal computations once within their existing code, then build around them. This again reflects students' difficulties in decomposing the problem around the tasks.

4.3 Advanced Techniques

Both accumulating intermediate data in parameters and reshaping the data into a list of sublists are advanced patterns that students had not seen in the course (and thus could not have retrieved). Only 1 student attempted to add a parameter for the running sublist sum. Only 3 attempted to reshape the data; none did so successfully. The coding sequences for the latter students suggests that as soon as students pulled out tasks to attempt to reshape the data, they proceeded into tinkering in and around these helpers. Students who tried this step (which would have been valuable had it worked) were clearly not thinking through focals, as data reshaping is not a computational task suggested in the problem statement (even though the problem hints at the flattened state of the data).

5. DISCUSSION AND FUTURE WORK

The study in this paper was motivated by an apparent underlying tension between HTDP templates and Rist's model of bottom-up programming. Rist's work suggests a cognitive process that students follow on new programming problems: write a core computation for a problem task (the focus), then augment the program to produce the data needed for the focal computation. The potential tension with HTDP lies in the sub-expressions that templates introduce: these provide a *context* into which students will place focal computations. That context could either help or interfere with students' thinking as they integrate focals into a larger program.

Our data suggest that students largely work through problem tasks: they write focal computations on the front elements of the input list, or create new functions for problem tasks. They often appear to retrieve plans, in the form of individual recursive functions, for individual tasks (such as summing a list). As such, our students often introduced focals as entire functions, not small expressions (as in Rist). This may have deferred students' entry into creation mode, shifting more burden to plan composition.

Our students struggled to compose plans: some failed to adjust the portion of the list being recurred over, others tried to perform two tasks with different output types (summing and building lists) within the same recursive traversal (rather than accumulating the sum or creating a helper). In both cases, our students used template expressions verbatim, rather than adjust them to the need of the computation at hand. Given that they had not seen programs that adjusted template code, this behavior is not particularly surprising. More generally, they lacked schemas for *composition*, instead retrieving insufficient, lower-level, plans.

Does this mean that templates, and the context they provide, interfere more than help students? Not necessarily. The recursive calls are still needed, even if on slightly different inputs in some (but not all!) cases. Students had simply not learned when to decompose problems into multiple instances of templates, and the templates failed to help them discover or resolve the issue. Given prior work on the importance of retrievable schemas, HTDP's templates fit squarely within known results on how people program.

Arguably, the key issue is that students are decomposing the problem *on the fly around code they have already written*. This arises whether students use HTDP (which prescribes the context) or bottom-up programming (in which students' existing code provides the context). If the prior context isn't well-suited to the problem at hand, students will struggle with composition. Decomposing the *problem* up front, into tasks that can be composed cleanly into a solution, should make the actual coding less error prone.

Can we teach students to effectively decompose problems? Both Rist's work and ours show that students think in terms of core program tasks. Decomposing problems (rather than code) is about grouping the tasks of a problem into chunks that can reasonably be handled together. What if we could teach students to use concrete examples to work out problem decompositions? In the case of *Adding Machine*, for example, a student could start with

```
(adding-machine (list 1 2 0 7 0 5 4 1 0 0 6))
```

then write that this should produce the same answer as

```
(list (+ 1 2) (+ 7) (+ 5 4 1))
```

Realizing this might suggest specific functions that a student could write to transform the first expression into the second. Something systematic such as this seems preferable to expecting students to just keep experimenting until their bottom-up process hits on a workable solution. Just as students currently internalize schemas for writing code, we might expect they can learn to internalize schemas for decomposing problems through concrete examples. The question at least warrants further study.

We suspect that some tension between Rist's model and HTDP arises from differences between functional and imperative programming. Deciding what Rist's model might mean within functional programming took considerable discussion among the authors. Rist's description of "bottom up" references code organization typical of imperative programs: variable declaration and initialization at the top, computation in the middle, and results and output on the bottom. Functional programs are organized differently: variables are initialized when functions are called, and outputs are typically composed from nested expressions within the middle of a function. For this project, we chose to interpret "bottom up" as "write contextual code after the focus". Functional programs also tend to decompose problems into several functions, whereas imperative solutions for CS1-level programs often live within a single procedure. This changes the decomposition patterns that students need for problems such as *Adding Machine*. This raises various questions for how cognitive behavior in creation mode might differ based on the (affordances of the) programming language at hand.

Acknowledgments.

We thank Joe Beck for letting us collect data in his course and Mike Clancy for pointing us to the *Adding Machine* problem. This research is partially funded by the US NSF under grant number CCF1116539.

6. REFERENCES

[1] M. Berland, T. Martin, T. Benton, C. P. Smith, and D. Davis. Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences*, 22(4):564–599, Oct 2013.

[2] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[3] P. L. Pirolli and J. R. Anderson. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie*, 39(2):240–272, 1985.

[4] P. L. Pirolli, J. R. Anderson, and R. G. Farrell. *Learning to program recursion*, pages 277–280. 1984.

[5] R. S. Rist. Schema creation in programming. *Cognitive Science*, pages 389–414, 1989.

[6] R. S. Rist. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Hum.-Comput. Interact.*, 6(1):1–46, Mar 1991.

[7] B. Shneiderman. Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences*, 5(2):123–143, June 1976.

[8] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984.

[9] J. C. Spohrer and E. Soloway. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, July 1986.

[10] J. C. Spohrer and E. Soloway. *Simulating Student Programmers*, pages 543–549. IJCAI '89. Morgan Kaufmann Publishers Inc., 1989.