Structural versus Pipeline Composition of Higher-Order Functions (Experience Report)

ELIJAH RIVERA, Brown University, USA
SHRIRAM KRISHNAMURTHI, Brown University, USA

In teaching students to program with compositions of higher-order functions, we have encountered a sharp distinction in the difficulty of problems as perceived by students. This distinction especially matters as growing numbers of programmers learn about functional programming for data processing. We have made initial progress on identifying this distinction, which appears counter-intuitive to some. We describe the phenomenon, provide some preliminary evidence of the difference in difficulty, and suggest consequences for functional programming pedagogy.

CCS Concepts: • **Software and its engineering → Software notations and tools**; • **Applied computing → Education**; • **Computing methodologies**;

Additional Key Words and Phrases: higher-order functions, composition, program structure, plans

ACM Reference Format:
Elijah Rivera and Shriram Krishnamurthi. 2022. Structural versus Pipeline Composition of Higher-Order Functions (Experience Report). *Proc. ACM Program. Lang.* 6, ICFP, Article 102 (August 2022), 14 pages. https://doi.org/10.1145/3547633

1 AN EDUCATION PROBLEM

For a few years now, a colleague at our institution has been running an introductory computing course—henceforth DC, short for "data centric"—in which students begin with functional programming (FP) over *tables*, following the philosophy of a recent CACM article on curriculum design [Krishnamurthi and Fisler 2020]. Similarly, Author 2 has taught an "accelerated" introductory course (henceforth AC), designed primarily for students with significant prior computing experience. Both courses use higher-order functions (HOFs), over tables and lists, respectively.

In 2021, Author 2 charged Author 1 with creating new problems for AC for students to solve using only HOFs. Creating problems to exercise a student's ability to compose HOFs proved to be a more challenging task than expected; Author 1 struggled to propose reasonable problems, with most appearing to be either too easy or too difficult. Worse, both authors struggled to articulate characteristics that distinguished these classes of problems. We then found that there had been similar struggles with problems in DC as well. However, nothing in the FP literature provided a hint as to what that might be.

Ultimately, we conjectured that the *nature of the problems* might be a contributor. This paper distills our experience as follows. It starts by presenting two patterns of composing HOFs and asks readers to answer some questions about them. It then presents both answers from experts and our own views on these patterns. Next, it presents results from a preliminary experiment to investigate these views. Finally, we discuss implications for curricular design in FP.

Authors' addresses: Elijah Rivera, eerivera@brown.edu, Brown University, Providence, RI, USA; Shriram Krishnamurthi, shriram@brown.edu, Brown University, Providence, RI, USA.

2475-1421/2022/8-ART102
https://doi.org/10.1145/3547633

2 PAPER CONTEXT

In this paper, we limit ourselves to HOFs over lists, and permit only these functions:

`map`	`(A → B) * List<A> → List<B>`	transforms
`andmap`	`(A → Bool) * List<A> → Bool`	conjoins while mapping
`ormap`	`(A → Bool) * List<A> → Bool`	disjoins while mapping
`sort`	`(A * A → Bool) * List<A> → List<A>`	sorts by predicate
`filter`	`(A → Bool) * List<A> → List<A>`	filters by predicate
`take-while`	`(A → Bool) * List<A> → List<A>`	takes while predicate

Notably, we exclude the variants of `fold`. We discuss why in section 8.3.

3 A PUZZLE FOR THE READER

Before continuing, we ask the reader to stop and perform the following exercise.

Consider the following two structures of composing higher-order functions ("funarg" is short for the functional parameter), where `L` is a list:

Type A `HOF_A(<some funarg>, HOF_B(<some funarg>, L))`
Type B `HOF_C((\inner.HOF_D(<some funarg>, inner)), L)`

(Ignore interleavings of these.)

Now answer the following questions:

(1) Which of these types would you consider "easier" for students to understand and use?
(2) How would you rate their relative expressive power in terms of problems they can solve? That is, how you would compare the set of problems that can be solved with the two structures?

Don't skim over these; they're central to the rest of the paper. Consider writing down your answers, so you can match against what you read later.

Commit to your answer before you continue!

---

4 EXPERT TESTIMONY

We asked a few people at our institution with FP experience, and got consistent answers from them. We then reached out to three well-known FP educators at three other (all different) institutions and posed the same questions. We carefully chose a person each closely affiliated with (and who has written a high-quality FP education book using) Haskell, OCaml, and Scheme, the venerable forebears of today's variety of languages. All three are also established researchers in their own right. We got essentially the same answers from them as we obtained internally:

(1) Type A is probably much simpler than Type B. The exact reasons differed but were similar: B potentially requires working with `lambda`s, while A corresponds to "pipeline" or a sequence of loops (which many consider simpler, though this was expressed with some skepticism).
(2) We did not obtain clear-cut answers to this question, suggesting a lack of certainty.

We are certainly curious to hear whether our readers had similar conclusions.

For the rest of this paper we will adopt different terminology. We henceforth refer to Type A as a *pipeline* composition, because it corresponds to `HOF_B L | HOF_A` (in Unix notation), and matches common functional, Unix, data science, and other data processing pipelines. We will call Type B *structural*, for reasons that will soon be clear.

5 REASONING ABOUT THE PUZZLE

Let's first consider the second question. To answer it, it helps to take the perspective of a type-driven synthesis engine [Feser et al. 2015] searching for possible functions that can be used in a spot.

There is, in fact, a very narrow range of compositions that can be written in the structural style. Observe that, for the introductory HOFs we consider, `inner` must be an *element* of `L`; therefore, the set of available HOFs that can take the place of `HOF_D` must be those that operate on the type of `L`'s content. Thus if `L` is itself a "flat" list, then there are *no* such functions in the novice's vocabulary. At this point the composition has to "bottom out" to use built-in, library, or custom functions, but there is no need to search further in the (restricted) space of HOFs.

This explains our choice of the terminology. In this style, at every level, a list-consuming HOF "peels off" one level of structure; therefore, the set of possible "next" functions can only be those that can accommodate what is left of the structure. Thus, the type structure of the original (list) input greatly limits the set of functions *and* the depth of nesting that is feasible.[1]

The pipeline form has no such restriction. The list input to `HOF_A` is not at all constrained in the way the funarg is in structural composition. In fact, `HOF_B` can even produce output that is "deeper" than the type of `L` (e.g., when `map` consumes a list and produces a list-of-lists). This means the set of possible functions that can go in each location is much larger, and many more combinations have to be considered. Furthermore, the pipeline can be made arbitrarily long (as any experienced Unix user knows), so without an a priori bound, the solution search can be unbounded.

In short, *our intutions* (but perhaps not yours!) *seem to be wrong*. However, this is an argument based on anecdotal information and a proposed theory. But *do* students actually find structural problems simpler than pipeline problems?

6 THE EXPERIMENT: STUDY DESIGN

To probe this issue experimentally, we conducted a study in AC (the course we had easiest access to). This section describes the study; section 7 analyzes the responses.

6.1 How to Assess?

It would be natural to look at student performance in the above classes as an indicator of the difficulty of these problems. However, this would almost certainly be misleading, for several reasons:

(1) At our institution (as at many others), students can drop a course after starting it (and perhaps re-take it later). Thus, students who struggled most may not be represented in the data.
(2) At our institution, there are multiple pathways to the degree. Thus students can choose to switch to a different course, again introducing a selection effect.
(3) Students can avail of help from TAs, who are especially likely to be helpful early in the semester. Looking at final performance does not reveal these interventions.
(4) Finally, when grades are at stake, some students may get illicit "help" from other students or other sources. These too would be difficult to identify from final answers.

Some of these phenomena can be mitigated by, e.g., close surveillance. We expressly chose not to follow this path because this can make students deeply uncomfortable and would hence adversely affect their performance. It can have an especially negative effect on students who feel they do not "belong" in computer science, thereby hurting diversity.

[1] One caveat is that there can be "hidden" levels of nesting. For instance, the original datum might be a list of strings, which suggests only one level of nesting, but—depending on the language—a string either is or can be exploded into a list of characters, resulting in another level of nesting.

The Planning Alternative. What we would like to see, unintrusively, is student "thinking" about how they might structure a solution. In particular, we are curious to see what they think about *before* producing a final artifact. To do this, we turned to the classic computing education field of program *planning*.

Driven by cognitive science, planning was introduced in the mid-1980s [Johnson and Soloway 1984] as a more concrete, and cognitively-grounded, version of "program design". Unfortunately, early negative results [Ebrahimi 1994; Johnson and Soloway 1984] made progress difficult. In the past decade, however, several papers [Achten 2021; Fisler et al. 2016; Simon 2013] have revived this field, with some [Achten 2021; Fisler 2014; Fisler et al. 2016; Seppälä et al. 2015] showing a particularly strong tie to thinking functionally and compositionally.

We therefore decided to ask students to present *plans* for a collection of problems, with a particular emphasis on using HOFs as the planning primitives. This framing is somewhat narrow, but has the benefit of being very concrete, and also reinforcing a pedagogic context we cared about (namely, making students build a vocabulary of functional operators, and get comfortable thinking compositionally in terms of them).

6.2 Study Context

We conducted our study in AC. At the point where we conducted the study, all students had been assigned to read the higher-order functions material in *How to Design Programs* [Felleisen et al. 2001] (HTDP). They had also completed an assignment where they had to write several functions *using only* HOFs, with explicit recursion forbidden. Even lacking intrinsic motivation, the latter, presumably, motivated students to complete the reading. They used the Advanced Student Language level [Findler et al. 2002] of Racket [Flatt and PLT 2010]. Though the student levels are not statically typed, they were required to read and write contracts-as-comments in the style of HTDP.

The students were familiar with all the functions listed in section 2 with the exception of `take-while`. The task gave them a definition and example of `take-while`. We intentionally added this function for three reasons: (a) to enrich the set of problems, (b) to confirm [Krishnamurthi and Fisler 2021]'s claim that students often have confusion between `take-while` and `filter`, and (c) to see whether the inclusion of an unfamiliar operator would cause significant problems. (It didn't!) These students were also familiar with `fold`, even though we leave it out of this study.

6.3 Problem Description

Our study had seven programming problems. Each of these problems was intended to simulate the kind of task that confronts many people in data processing: given a concrete input, they have a sense of what kind of output they want at the end. The task is then to generalize this process into a computation that can be applied to any dataset.

For each problem, we needed a way of presenting it to the students. We chose to not use a verbal description because, despite several tries, we were unable to find formulations that did not effectively give away the *solution* structure to lesser or greater extents. Verbal descriptions have the additional detriment of being difficult to standardize, both for length and language complexity. For any set of verbal descriptions, it could be argued that differences in the *statement* of the problems is what tripped up a student, and not the *essence* of a set of problems. We sought to mitigate this effect by selecting a more standardizable problem representation.

We therefore described each problem through three input-output pairs.[2] Our examples use the syntax of Lisp, but should be trivial to adapt to any language with list data structures.

[2]We do not claim this process of problem description is ideal. See section 8.2.

Our work was done in a purely-functional setting: students were in the pure portion of HTDP and were required to program without side-effects, which was enforced by their language DrRacket level (as noted in section 6.2). The work in this paper fundamentally depends on purity. In the presence of state, a `map` or `filter` is no longer limited to working on each element independnetly and can instead exhibit `fold`-like aspects, which would put it outside the scope of our study.

6.4 Problem Design and Solutions

All seven problems are shown in fig. 1. Three problems were designed to have a structural solution (S1–S3). S1 composes `map` and `filter`, S2 composes `map` and `take-while`, and S3 `filter` and `andmap`. Three (P1–P3) were designed for pipeline solutions. Their solutions use the *same* pairs of HOFs as S1–S3 respectively: e.g., P1 composes `map` and `filter`. Finally, one problem (I1) asked them to duplicate the input list, which is impossible to do with the given operations. The intended solutions are shown in fig. 2.

6.5 Response Task

Given the three examples, students were told, "Your job is to determine a way to combine higher-order functions (HOFs) to produce **all** of the examples [...]. For any HOF used, the function parameter may be a built-in function or a custom lambda. The function parameter may also be a lambda that invokes another HOF from the list above." Students were given 5 days to respond to the set of questions with no limit on how many hours they spent.

In response to the question "How would you combine the available higher-order functions to do this?", students could answer one of

- I see how to produce these examples using a combination of higher-order functions (I See)
- I don't think these examples are possible to produce using the given higher-order functions (Not Poss)
- I don't know (idk)

(The short codes in parentheses are how we will refer to these responses later in the paper.) Each selection came with a corresponding follow-up question, which students were asked to answer narratively in a text box. The follow-up questions were:

I See: How would you combine the available higher-order functions to do this?
Not Poss: Why do you think this is impossible?
idk: What do you find difficult about these examples?

6.6 Problem Order

Students were shown the problems in random order to minimize sequence, order, and carryover effects [Salkind 2010]. However, students were always shown problem S1 first to start off with what we hoped would be an easy problem (as it appears to have been). We especially wanted to avoid I1 appearing first—which it could have if we had completely randomized the order—which might have been rather confusing (and perhaps also off-putting). Showing S1 first avoids that.

7 THE EXPERIMENT: ANALYSIS

Now we study student responses to these questions.

S1.

```
(list (list 6 0) (list 5 1) (list 9 2 5)) → (list (list 6 0) (list 1) (list 9 2))
(list (list 5) (list 4 5)) → (list empty (list 4))
(list (list 6 5 7) (list 6 0 5) (list 9 1) (list 5 5)) →
  (list (list 6 7) (list 6 0) (list 9 1) empty)
```

S2.

```
(list (list "my" "account" "earns" "interest")
      (list "her" "dad" "took" "interest" "in" "her" "work")
      (list "i" "lost" "interest" "so" "i" "left")) →
    (list (list "my" "account" "earns")
          (list "her" "dad" "took")
          (list "i" "lost"))
(list (list "the" "old" "park" "at" "the" "town" "center")
      (list "park" "there" "and" "walk")) →
    (list (list "the old") empty)
(list (list "can" "i" "play" "if" "i" "finish" "work")
      (list "i" "am" "so" "excited")
      (list "my" "class" "play" "is" "my" "next" "event")) →
    (list (list "can" "i")
          (list "i" "am" "so" "excited")
          (list "my" "class"))
```

S3.

```
(list (list -6 1 2) (list 1 4) (list 2 6 9)) → (list (list 1 4) (list 2 6 9))
(list (list 7 -2 3 -5) (list -4 -1)) → empty
(list (list 8 5 0) (list -6) empty (list 9 1)) → (list (list 8 5 0) empty (list 9 1))
```

P1.

```
(list -40 212 32 0) → (list -40 100 0)
(list 32 0 212 32) → (list 0 100 0)
(list -40 -40 32 -40 0 212) → (list -40 -40 0 -40 100)
```

P2.

```
(list "red" "amber" "green" "blue" "brown") → (list 6 10 10)
(list "orange" "purple" "pink" "gray" "purple") → (list 12 12)
(list "teal" "periwinkle" "pink") → empty
```

P3.

```
(list -7 1 -4 -8 2) → #false
(list -3 8 -3 6 2 -5 4) → #true
(list -3 -6 -5) → #true
```

I1.

```
(list 1 5 -5) → (list 1 5 -5 1 5 -5)
(list 0) → (list 0 0)
empty → empty
```

Fig. 1. All problems (originally presented on multiple lines; spacing reduced to fit on page)

S1.

```
(define (question1 outer-list)
  (map (lambda (inner-list)
         (filter (lambda (elem) (not (= elem 5)))
                 inner-list))
       outer-list))
```

S2.

```
(define (question2 outer-list word)
  (map (lambda (inner-list)
         (take-while (lambda (elem) (not (equal? elem word)))
                     inner-list))
       outer-list))
```

S3.

```
(define (question3 outer-list)
  (filter (lambda (inner-list)
            (andmap (lambda (elem) (> elem 0))
                    inner-list))
          outer-list))
```

P1.

```
(define (f-to-c f) (* (- f 32) (/ 5 9)))

(define (question4 outer-list)
  (map f-to-c
       (filter (lambda (elem) (not (zero? elem)))
               outer-list)))
```

P2.

```
(define (question5 outer-list)
  (map (lambda (elem) (* 2 (string-length elem)))
       (take-while (lambda (elem) (not (= (string-length elem) 4)))
                   outer-list)))
```

P3.

```
(define (question6-intended outer-list)
  (andmap even?
          (filter positive?
                  outer-list)))


(define (question6-alternate outer-list)
  (ormap (lambda (elem) (= elem -3))
         outer-list))
```

Fig. 2. Solutions

7.1 Numeric Student Responses

Numerically, student answers were as follows:

Problem Type	I See	Not Poss	IDK
S1	40	0	0
S2	36	1	3
S3	40	0	0
P1	28	*8*	*4*
P2	16	*8*	*16*
P3	30	*2*	*8*
I1	5	29	6

We can see significantly greater lack of certainty when it comes to the pipeline problems. The impossible problem was included to serve as an "attention test", and indeed only a small number of students claimed to be able to solve it.

Of the pipeline problems, note that P3 seems qualitatively a little different from P1 and especially from P2. P3 highlights a difficulty in the construction of pipeline problems: it is possible to create problems with simpler solutions than intended. In this case, while P3 had the intended compositional solution `andmap(even?, filter(positive?, L))`, it can also be solved with just a single `ormap` or `andmap` (e.g., `(ormap (\x -> x == -3) L)`). The textual responses (studied below) show that it is these alternative solutions, not the recognition of the pipeline solution, that contributed to better scores than the other two pipeline problems.

This failure in problem creation was unfortunate for our intended study, but in turn highlighted how unforeseen challenges in problem creation can also impact intended classroom learning. We return to other aspects of this issue in section 8.1.

7.2 Narrative Student Reponses

Of course, it is not enough to just look at the numeric results. We should also see why they chose their answers; indeed, these prove to be instructive.

S1. 37 students provided seemingly correct answers. Two missed an explicit `map`, and one an explicit `filter`.

S2. Of the 36 I See answers, 31 had seemingly correct compositions. Three were missing an explicit `map`. Only two were incorrect, using `filter` instead of `take-while`. It is worth noting that confusion between `filter` and `take-while` has been noticed by other researchers too [Krishnamurthi and Fisler 2021], but to much greater degrees than we see.

The one Not Poss inferred a much richer problem—that elements were kept so long as they were not found in later sub-lists—thereby effectively requiring a `fold`, which had not been provided (though they used the phrase "taken while").

Of the three IDK students, all three were on the right track; two indicated `map+take-while` but inferred a harder problem, while the third focused on `map+filter` (recalling an earlier homework problem) and could not see how to solve it.

S3. 31 had seemingly correct compositions with `filter` and `andmap` or `ormap`. Four were missing an explicit `andmap` or `ormap`. Five were incorrect: three had an extra outer `map`, one assumed `map` can behave like `fold`, and one used `take-while` instead of `filter`.

Structural Summary. In short, on the structural problems, the vast majority of students did either well enough or perfectly, and there were very few errors and only for some problems. That is, they chose the right answer *with the correct reasoning*.

P1. Only 28 students chose I See. Of these, 22 are seemingly correct, three are incorrect, and the others are ambiguous. The responses included entries like:

- "*Whew! This one took a while to figure out. It's a three step process(at least the way I do it)*". Their first step is a `map` that converts 32s to 0s and 212s to 100s. Their second is a `foldr` (not on our list) to delete zeroes to the right of a 100. Their third is a `foldr` to eliminate zeroes to the immediate left of another 0. They explain the use of `foldr` and provide a detailed example.
- A `map` "*with a unique function that utilizes filter to eliminate repeat results*".
- "*I believe this should be take-while and map, with a helper included in map*"

and so on. The Not Poss answers included responses like "*andmap and ormap return booleans; map returns the same length; the other three HOFs can't add new elements*" and "*It's both altering and filtering the elements from the list*". The idk responses made clear that they could not find a pattern.

P2. Only 16 students chose I See. Of these, 10 were seemingly correct, one missed an explicit `take-while`, while five were incorrect. Students in Not Poss had similar answers as for P1 (including some identical quotes), along with claims that they could not see a pattern. The idk responses again said they could not see the pattern.

P3. Of the 30 I See responses, 27 saw the simpler pattern previously mentioned (a single `andmap` or `ormap`), two had seemingly correct compositions, and one was incorrect. One of the Not Poss made a type-and-value argument, while the other (correctly) assumed an overly simple pattern and argued it did not require hofs. The idk responses again did not find the pattern, except one who may have but said "*It's either andmap or ormap, but I don't know how to determine which is being used here*".

Pipeline Summary. In short, even the "correct" answers for P1–P3 included incorrect reasons. Lacking an intermediate step, there was considerable difficulty determining how the input had been transformed into the output. We feel safe in concluding that this set of problems, for these students, truly was harder.

I1. Three of the I See answers were not decipherable to the authors. One explicitly assumed a "*duplicate*" function. Finally, one described (without using the name, but referencing "*accumulator*") a `fold`-style function. The Not Poss arguments all provided an argument in terms of the size and type properties (which we discuss more in section 10.3). The idk answers were essentially the same.

8 DISCUSSION

The preceding analysis will likely inspire certain questions or challenges from the reader, which we try to anticipate here.

8.1 Are These Just Trick Questions?

We see that there seems to be considerably more confusion and ambiguity with the pipeline problems. A reader's first reaction might be to think that the whole setup is unfair: the pipeline problems might seem to be "trick questions" and coming up with a solution is therefore really a question of whether the reader sees the trick. In response, we offer two arguments.

First, and returning to the phenomena in section 1, they did not appear that way to Author 2 when constructing the problem—because he already knew the solution. We argue, from personal experience that we have not yet tried to scientifically evaluate, that knowing the solution structure can create a giant blind spot that obscures the difficulty of problems.

Second, we don't even disagree with this characterization; indeed, it is part of our point! Structural problems are deeply constrained. The *same* pairs of functions, in the pipeline case, lead to vastly many more options, making it tricky for students to see what compositions might get them to a solution. Put differently, the student-as-synthesizer has a much bigger space to traverse. In particular, it's important to remember that the *same* students who are able to solve structural problems—often by describing clear scaffolding aids in their responses—struggle to do so in the pipeline cases.

Despite these issues, pipeline problems cannot simply be skirted, especially as they may represent many common data processing tasks. We therefore discuss pedagogy in section 10.2.

8.2 Isn't This a Bad Choice of Representation?

Our input-output representation of a problem, with no corresponding textual description or "real-world" meaning, is almost certainly overly restrictive. Would students have had this much difficulty if presented with a good textual specification?

(1) As we have noted earlier in this paper, this style of presenting problems helps us circumvent numerous factors that would have been major confounds. Thus, it is still valuable from an experimental perspective, though we must allow that it may subtly advantage one style of problem over another.
(2) More importantly, remember that we were not creating an artificial activity in vitro. Rather, we were responding to an already observed anecdotal problem—which had been observed in a setting with traditional verbal problem descriptions.

Nevertheless, it is worth asking what a textual description might look like. Before we settled on this presentation, we tried numerous ways to express the problems verbally. What we found was that our descriptions were either too obscure (hardly better than the examples, and perhaps even more confusing for their lack of concreteness) or effectively *gave away the solution strategy*. Since our goal was to see whether students were able to find the compositions, the latter approach defeated the objective. We were, however, unable to find a happy medium between these extremes. We believe finding a strategy between these extremes is an important issue (see section 10.2).

8.3 What About `fold`?

We intentionally left the variants of `fold` out of our study due to the universality of that function, making responses harder to evaluate. In this, we were inspired by other authors [Krishnamurthi and Fisler 2021], who have also found it confusing to include `fold` because it makes student responses more ambiguous: in particular, a student who is either overly clever or lazy could write `fold` in response to all questions, which would not be wrong but also not be informative. In addition, we have found that students at the stage we are studying generally have a much poorer understanding of `fold` than of the history-less HOFs, often possessing only a mechanical understanding. Nevertheless, we do see some evidence of students recognizing uses for `fold`, and many more might have recognized it but eschewed it because it was left out of our set.

8.4 What Are the Confounding Factors?

What we have presented is mostly anecdotal and narrowly scoped; to make up for the narrow scoping, we have gathered some hard data showing that, at least in this setting, there is evidence of a problem. However, there are numerous confounding factors that may cause this to be a strictly local phenomenon and/or an artifact of some other difficulty. For instance:

- As previously discussed, the very way we have stated the problem is itself potentially problematic; the stark input-output modality turns this into a pattern-finding activity. Thus, much

more work is needed to address these issues and create a credible, generalizable, and hence robust research result.
- Some of the AC problems that were found to be difficult actually relied on `fold` or some other such history-sensitive operation; that alone could have been the cause of why staff found them more difficult. (In mitigation, the pipeline problems in this paper do not require `fold`, and yet create difficulty, especially once we consider the narrative answers.)
- Some problems may have lurking domain knowledge: for instance, P1 above clearly relies on students recognizing familiar Fahrenheit/Celsius patterns, but the narrative responses suggest that a handful of them failed to do so. (In mitigation, P2—which does not have a similar dependence—does no better, and indeed much worse.)
- Recall that our work was motivated in section 1 by two courses, one of which (DC) was based on programming functionally over tables rather than lists. We have to therefore be open to the possibility that tabular datatypes introduce additional difficulties that we have not considered, some of which could even be dominant. (In mitigation, we see here that lists are problematic enough.)
- Our students programmed in Racket. It is possible that explicit and checked types, automated currying, or writing operations left-to-right (using pipelining notations) instead of right-to-left (through nested composition), could alter how students program, and even impact their plans (which is what this paper studied).

At the same time, the *positive numbers given above may well be an over-count.* The AC students had much more programming experience than those of DC, so they may have been able to work their way through their difficulties. The AC students may also be more mathematically sophisticated. Lacking these advantages, the DC students—who are probably much more representative of novice programmers broadly—may do much more poorly, either overall or only on pipeline problems. We return to discussing student populations in section 10.4.

9 RELATED WORK

Our primary related work is [Krishnamurthi and Fisler 2021]. That work showed both that students can learn HOFs well (which is an implicit prerequisite of this work), and established the precedent of using input-output pairs as an instrument to study HOF problems.

The initial motivation for this paper came about when we tried to extend that paper, applying their methodology to *compositions* of HOFs. We found it surprisingly difficult to create input-output pairs representing some compositions. We reflected on our difficulty, and then wedded it to the much richer, longer-term observations about courses, as discussed in section 1.

A companion paper [Rivera et al. 2022] studies the question of plans for composition problems in much more detail, with a rigorous methodology to analyze student responses. That paper exploits the insights of this one by focusing on structural problems, since pipeline problems were felt to be too difficult. That paper also introduces the idea of using a block-based syntax to have students describe their plans.

Identifying when to derive the program structure from the datatype is a known open question in HTDP as well [Castro and Fisler 2020]. When does one use structural versus generative recursion? To be overly reductive, HTDP's answer essentially boils down to "attempt to use structural, and if it doesn't work well, then you know to use generative". One could imagine giving similar advice for pipeline versus structural composition (section 10.1), but it might be possible to do more with scaffolds (section 10.2).

10 PEDAGOGIC DIRECTIONS

Next we discuss some of the pedagogic consderations induced by this work.

10.1 A Pedagogic Progression

We believe there is a strong parallel between the distinction this paper raises and that between "structural" and "generative" recursion described by HTDP. Structural recursion follows the structure of the datum: each sub-problem is given by the data definition. Structurally recursive problems therefore have three advantages:

- At a mechanical level, termination over freely-generated terms is guaranteed, because each recursive call "peels off" a layer.
- At a solution level, the structure of the datum suggests a useful starting point for the structure of the solution.
- There are commonalities between problems over the same datatype, so students can transfer [Thorndike and Woolworth 1901] some of their knowledge.

In contrast, in generative recursion, recursive calls computationally generate new sub-problems. Therefore, at a mechanical level termination is not guaranteed (the sub-problem may be the same size or even bigger, and complex termination criteria may be necessary); at a solution level, they can require an "a ha!"; consequently, there is little transfer between problems, each of which requires different insights.

Our work is strongly inspired by this distinction in HTDP. Indeed, we use the name "structural" due to this inspiration. We chose the name "pipeline" rather than "generative" (a) because the term is already well-established in data processing and data science, (b) to reduce confusion with the pair of terms in HTDP, since our distinction is between styles of *composition*, not *recursion*.[3]

HTDP makes (and implements) the case that education in recursion should start solely with structural problems. It argues that only after students are comfortable with the mechanics of structural recursion should pedagogy turn to generative problems. We see strong parallels here. To be sure, the set of problems we can solve structurally seems to be quite limited (perhaps even more so than with structural recursion). Nevertheless, it is rich enough to express some useful and perhaps interesting problems (e.g., we can phrase problems in terms of tasks recognizable from games like Scrabble). Anyway, interesting problems that students cannot solve can be counterproductive!

10.2 Scaffolding Pipeline Problems

We have spent some time reflecting on what makes pipeline problems appear harder. We conjecture that one source of difficulty is that the student cannot see the values at intermediate stages during the composition. Absent this visibility, the set of possible transformations is simply too great. We conjecture that in contrast, the educators who assign these problems do not suffer from this either due to experience, resulting in an expert blind spot [Nathan et al. 2001], or due to the simple expedient of having worked backward from an intended solution.

At the same time, educators cannot shirk pipeline problems entirely. Many interesting, and real-world, data processing tasks are pipelines.

Therefore, when introducing them early in the curriculum, we recommend that educators *scaffold* them. One way is to word the problem statement to give very explicit directions, but this runs the risk of essentially giving away the solution, thereby eliminating a learning opportunity. Another is to provide examples of intermediate results from which students can infer the pipeline stages.

[3]It is worth considering what solutions would look like if they were written with explicit recursion rather than by composing HOFs: would their recursive solutions be structural or generative? It is currently unclear to us the extent to which these two notions might be related.

In some cases it may be useful to suggest intermediate *types* alone, but in cases where the type does not change—such as the table-transforming programs of DC—it is unclear how much this will help students and how much it will frustate them. Our hope is that the dichotomy introduced in this paper will lead to much more research on programming methods, which can help educators analyze problems and determine which may need more scaffolding than others.

10.3 Problem-Solving Strategies

There is a small but significant literature on the problem-solving strategies employed by programmers. Early studies argued for different "forward" and "backward" search strategies employed by novices versus experts, though these did not survive subsequent analysis. These ideas were summarized and extended by Rist [1989], who found that programmers recall *plan schemas* to aid in solution, and attributed some differences in performance to knowledge and recall of these schemas.

How do these results play out in our setting? Most of this work rarely considered data-rich problems or, in particular, problems with rich data *structure*. In contrast, our students had been shown a table (Figure 1 of [Krishnamurthi and Fisler 2021]) characterizing high-level properties of most of the HOFs we use: • output type; • for list outputs, output element type relative to input; • output length relative to input; • output order relative to input; • which/how many elements of input determine an output element; and • type of operation consumed by the HOF. Our findings show that students clearly exploit various type- and structural-properties of HOFs in problem decomposition; this is especially evident from problems like I1, where students use these properties to argue why a solution is impossible. Thus, it would be interesting to study the role of semantic properties of high-level operations (such as HOFs) in planning. It would also be interesting to consider how students would fare in an *output*-driven setting [Gibbons 2021] as opposed to one driven by inputs (as described by HTDP).

10.4 From Experience to Research

In computing education research, it is increasingly common to conduct multi-institutional, multi-national (MIMN) studies. This helps account for many variables, and identify true trends across student populations. To this, we would add ML: multi-lingual. We are therefore offering to host a MIMNML study, and invite educators who are interested to reach out to us!

ACKNOWLEDGMENTS

We thank Matthias Felleisen and Kathi Fisler for useful conversations and feedback. We appreciate the detailed reading by the reviewers. We especially appreciate our shepherd, Dan Grossman, who went well beyond the call of duty to help improve this paper. This work is partially supported by the US National Science Foundation.

REFERENCES

Peter Achten. 2021. Segments: An alternative rainfall problem. *Journal of Functional Programming* 31 (2021). https://doi.org/10.1017/S0956796821000216

Francisco Enrique Vicente Castro and Kathi Fisler. 2020. *Qualitative Analyses of Movements Between Task-Level and Code-Level Thinking of Novice Programmers.* Association for Computing Machinery, New York, NY, USA, 487–493. https://doi.org/10.1145/3328778.3366847

Alireza Ebrahimi. 1994. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies* 41, 4 (1994), 457–480. https://doi.org/10.1006/ijhc.1994.1069

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

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239. https://doi.org/10.1145/2813885.2737977

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming* 12, 2 (2002), 159–182. https://doi.org/10.1017/S0956796801004208

Kathi Fisler. 2014. The recurring rainfall problem. In *SIGCSE International Computing Education Research Conference*. 35–42. https://doi.org/10.1145/2632320.2632346

Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *ACM Technical Symposium on Computer Science Education*. https://doi.org/10.1145/2839509.2844556

Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR2010-1. PLT Inc. http://racket-lang.org/tr1/.

Jeremy Gibbons. 2021. How to design co-programs. *Journal of Functional Programming* 31 (2021). https://doi.org/10.1017/S0956796821000113

W Lewis Johnson and Elliot Soloway. 1984. Intention-based diagnosis of programming errors. In *National Conference on Artificial Intelligence*.

Shriram Krishnamurthi and Kathi Fisler. 2020. Data-Centricity: A Challenge and Opportunity for Computing Education. *Commun. ACM* 63, 8 (2020). https://doi.org/10.1145/3408056

Shriram Krishnamurthi and Kathi Fisler. 2021. Developing Behavioral Concepts of Higher-Order Functions. In *SIGCSE International Computing Education Research Conference*. https://doi.org/10.1145/3446871.3469739

Mitchell J Nathan, Kenneth R Koedinger, Martha W Alibali, et al. 2001. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *International Conference on Cognitive Science*.

Robert S Rist. 1989. Schema creation in programming. *Cognitive Science* 13, 3 (1989), 389–414. https://doi.org/10.1016/0364-0213(89)90018-9

Elijah Rivera, Shriram Krishnamurthi, and Robert Goldstone. 2022. Plan Composition Using Higher-Order Functions. In *SIGCSE International Computing Education Research Conference*. https://doi.org/10.1145/3501385.3543965

Neil J Salkind. 2010. *Encyclopedia of Research Design*. SAGE Publishing. https://doi.org/10.4135/9781412961288 Entry for "Sequence Effects".

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

Simon. 2013. Soloway's Rainfall Problem Has Become Harder. In *Learning and Teaching in Computing and Enginering (LaTiCE)*. IEEE Computer Society, Los Alamitos, CA, USA, 130–135. https://doi.org/10.1109/LaTiCE.2013.44

Edward L. Thorndike and Robert S. Woolworth. 1901. The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review* 8 (1901). https://doi.org/10.1037/h0074898