Developing Behavioral Concepts of Higher-Order Functions

Shriram Krishnamurthi
Brown University
Providence, RI, USA

Kathi Fisler
Brown University
Providence, RI, USA

ABSTRACT

Motivation. Higher-order functions are a standard and increasingly central component in many kinds of modern programming, including data science and Web development. Yet little research has been devoted to student learning or understanding of this topic.

Objectives. We conducted formative research on how well students are able to correlate higher-order functions with their input–output behavior. We also wanted to evaluate a variety of techniques for assessing their understanding.

Method. We created a series of instruments in which students were given either concrete input/output examples or abstracted diagrams of list transformations. Students were asked to cluster or classify these examples by their behavior, sometimes against a concrete list of higher-order functions and sometimes free-form. We administered these over the course of a month, and then once again three months later.

Results. We find that students initially have several difficulties with clustering higher-order function examples. With different instruments, we find that students are later able to do quite well, largely avoiding large-scale errors but making several small-scale ones. We also find some evidence of growth in their thinking about these operations. We also find weaknesses in the nature and order of techniques we used.

Discussion. Higher-order functions deserve far more attention than they have been paid in the literature on programming education. Their increasing use in several important domains makes this need critical. Our proposed methods for conducting such research are another contribution of this work. Our findings and methods should also be relevant for exploring how students understand libraries and APIs.

CCS CONCEPTS

• **Applied computing → Education**.

KEYWORDS

higher-order functions, behavior, clustering, classification

ACM Reference Format:
Shriram Krishnamurthi and Kathi Fisler. 2021. Developing Behavioral Concepts of Higher-Order Functions. In *Proceedings of the 14th ACM Conference on International Computing Education Research (ICER 2021), August 16–19, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3446871.3469739

1 INTRODUCTION

Higher-order functions (HOFs) are everywhere in programming. What began as a niche feature in functional languages, like Lisp, is now ubiquitous. Java, starting with Java 8, added them and corresponding datatypes (i.e., `Streams`), so Java developers can (and do) write code like

```
listOfStrings.stream()
  .filter(s -> s.contains("CS"))
  .collect(Collectors.toList)
```

Web developers routinely write code like

```
$("ul").filter(
 function() { return $("li", this).length == 2; })
  .css("color", "red")
```

in jQuery, a popular JavaScript library. Python programmers using Pandas, an extremely popular data-analysis library, also routinely write

```
df['withTax'] = df['price'].apply(lambda x:x*1.05)
```

R programmers in data analysis frequently write

```
Reduce(f = "+", x = 1:6, accumulate = FALSE)
```

All these use HOFs (`filter`, `collect`, `apply`, `Reduce`).

A bit of terminology is in order: there are two related concepts here, *anonymous* and *higher-order* functions. The former (usually called "lambda"s in many languages, most recently even Excel [16]) create function-values without names. The latter are functions that accept function-values. Naturally, they have a close symbiosis: the former create values that the latter can accept. However, HOFs can exist without anonymous functions. Indeed, some learning progressions pass *named* functions to HOFs before teaching anonymous functions. Our focus in this paper is on HOFs, irrespective of how the functions that are being passed to them were created.

There are many purposes for which one can use HOFs, and their pedagogy must match these purposes. Our focus here is on HOFs for *processing data.* This focus is shared in several quarters. The examples above from R and Python are both taken from data-processing code. Hadley Wickham, Chief Scientist of RStudio and an influential data science educator, devotes a chapter of *Advanced R* [29] to "Function Operators", and has begun to create videos [30] to introduce HOFs to R programmers. Another famous example of HOFs in large-scale data processing is the MapReduce operator [5], which was directly inspired by the standard HOFs `map` and `reduce`, and consumes "functions" that implement the mapping and reducing steps. The growing importance of data science and its use in machine learning therefore highlights the importance and urgency of quality HOF pedagogic methods for students. This ubiquitous need has not been matched by pedagogy. There is little literature on how to effectively teach with HOFs. Even introductory books that have significant coverage of them (e.g., [1, 8, 26]), have not been evaluated for their effectiveness in this area.

For decades, the authors of this paper have taught HOFs by having students write data-processing traversals as recursive functions over lists (without HOFs), then pointing out the similarities in the functions for similar tasks, and abstracting over the similar traversals to define HOFs from scratch. From this they proceed to showing the collection of HOFs built into the programming language being taught, and then have students define and use HOFs during the rest of the course. This approach is based on the philosophy in the Abstraction section of *How to Design Programs* [8]. In this approach, fundamentally, HOFs arise as *abstractions of code.*

Selecting and applying HOFs in practice, however, starts not from existing traversal code, but from a description of desired program behavior. This description could be in prose or in the form of an input/output example (or test case). Thus, to understand whether our default pedagogic technique is working, we need to study whether learning about HOFs as abstractions of code helps students perceive them as *abstractions of behavior* as defined by features such as input/output types and the relationships between input lists and the lists or values produced as output.

This paper reports on a multi-stage study in which we gave students several concrete input/output examples of operations on lists and asked them to associate them with behaviors supported by the HOFs described in section 4. Our first two studies asked students to *cluster* the examples by their behaviors and name the clusters. Our last three studies asked students to *classify* or label the same examples with specific HOFs that matched each one. The first four studies occurred just after the students were exposed to HOFs through the default (code abstraction) pedagogy. The fifth occurred three months later, after students had used the HOFs on multiple assignments. Our analysis centers on two research questions:

RQ 1. *Which behavioral features of* HOF*s are students attuned to after learning about them as code abstractions and writing short programs with them?*

RQ 2. *Which behavioral patterns of list transformations do students correctly associate with the names of* HOF*s that exhibit corresponding patterns?*

We analyze these by asking students to both cluster and classify examples of function behavior, presented textually and diagrammatically. We then compare their annotations against a ground-truth defined by us. This methodology, which has broad applicability, is itself a contribution of this work. We find that they successfully distinguish some features better than others. Section 8 provides several reflections on this work.

2 THEORETICAL BASIS

In asking students to label examples, cluster examples, and name clusters, we are asking students to form *abstractions.* Students had already worked with HOFs as abstractions over similar programs, but were now asked to work with them as abstractions over input/output behavior.

Theories of abstraction formation build on theories of *perception* [14, 15]: one must identify features of a collection of samples and discern which are important prior to abstracting over common features. In the case of HOFs from an input/output perspective, important features include the types, lengths, and relative order across inputs and outputs. Figure 1 summaries these features in detail for the HOFs used in this study. The textbook that students read to initially learn about HOFs (before our study started) makes references to these features, but did not contrast them as sharply as in the table (which we provided to students prior to the third stage of our study). Our study explores feature identification through the labels that students ascribe to examples.

The key behavioral features we have identified about HOFs are *relational*, in that they capture relationships between the inputs and outputs of a computation. Gentner and Kurtz [13] proposed *relational categories* as a fundamental form of abstraction, separate from other forms based on shared structure of objects (such as from common pieces of code). The *category status hypothesis* proposes that relational categories aid in cognitive retrieval of structurally-relevant knowledge, based on tasks involving labeling, summarizing, and describing materials [20]. Our study employs similar tasks as a first step towards developing relational categories.

While theories of how programmers develop schemas [21, 24] might also apply to our work, we believe abstraction-formation theories are more relevant. Schemas are partially-instantiated patterns of code (or pseudocode). They would be relevant to this study if we were asking students to implement the underlying traversals many times and (mentally) retrieve the patterns of those traversals (as with code abstraction). Here, we are more concerned with students retrieving the name of an appropriate HOF based on input/output behavior. The whole point of using an HOF is that the common code structure does not appear explicitly in the resulting program. While schemas might get developed along the way, development of the abstraction to a named HOF is our overarching goal.

3 RELATED WORK

Though there are numerous sources that *teach* HOFs, we are not aware of literature on students' *understanding* of them. Similarly, we have not found prior work on how students learn or perceive relationships between code abstractions and behavioral abstractions.

There have been studies that look at students' selection and use of HOFs in the context of plan composition. Fisler's cross-language study of the Rainfall problem [9] reported on students' successful use of `filter` (and occasionally `fold`). Fisler et al.'s related study [10] included students who were learning OCaml: they successfully used both `filter` and `take` on the study problems.

Design patterns abstract over and descriptively name common programming tasks [4, 12]. Relative to this project, we view design patterns as being similar to schemas (discussed in section 2), in that the goal is to learn a multi-line (or expression) code pattern rather than the name of a single built-in operation with a specific input/output behavior.

Selecting functions based on desired behavior arises when working with APIs. The API context is a bit different from the HOF one:

Functions\Criteria	OT	OET/I	OL/I	OO/I	WHE/I	Op Type
`map`	list	can differ	same	same	1	`A -> B`
`filter`	list	same	<=	same	1	`A -> Bool`
`take-while`	list	same	<=	same	1 *until* criterion, then 0	`A -> Bool`
`ormap`	bool	-N/A-	const 1	-N/A-	1 *until* `true`, then 0	`A -> Bool`
`fold`	any	can differ	can differ	any	prefix/suffix	`A B -> B`

where the column headers are given by the following legend:

Abbreviation	Expansion/Meaning
OT	Output type
OET/I	For list outputs, output element type relative to input
OL/I	Output length relative to input
OO/I	Output order relative to input
WHE/I	Which/how many elements of input determine an output element
Op Type	Type of operation consumed by the HOF

Figure 1: HOF Feature Summary

APIs are often stateful (so types are insufficient summaries of behavior) and they support domain concepts beyond what the programmer is already working with for themselves (e.g., access to a cloud service vs. extracting items from a list). Programmers often use APIs to extend the functionality of their programs, rather than to write the same program more succinctly (as with HOFs). Documentation and instruction are nevertheless key instructional tools in both contexts, making findings on how programmers learn from API documentation potentially relevant to our work.

Duala-Ekoko and Robillard's study of the questions that experienced programmers ask while learning APIs [6] shows that programmers often navigate API documentation based on types and keywords (which resemble the labels we ask students to construct). Thayer et al.'s theory of robust API knowledge [25] synthesizes many prior studies in discussing the importance of understanding API usage patterns, which include input/output examples. In API studies, developers typically have access to documentation that contains types, examples, and behavioral descriptions. We provided documentation only after the first 2 stages of our study, and even then it was in a modified form of a table summarizing types and features (fig. 1), rather than the richer behavioral descriptions found in API documentation. This was by design in our study, so we could explore how students' prior use of the code underlying an HOF would lead to understanding of input/output examples.

4 A SMALL GLOSSARY OF HIGHER-ORDER FUNCTIONS

We assume that the reader has a basic familiarity with the standard HOFs. Nevertheless, fig. 1 provides a summary of the key HOFs we use in this paper (and their distinguishing features).

Due to the point in the curriculum where we conducted these studies, we focused solely on functions over lists. `map` applies its function argument to every element of a list. `filter` applies its function argument, which should return a Boolean, to every element of a list, and uses the Boolean to decide whether to retain or drop that element in the result. `take-while` is subtly different from `filter`: it retains the prefix for which the Boolean is true, and at the first false value, drops the suffix. `ormap` also takes a Boolean-generating function, which it applies to every element, and computes the disjunction of the produced Booleans; it short-circuits once it gets a first non-false value. `fold` is a generalized accumulating loop.

Why these functions? We chose `map`, `filter`, and `fold` because they are three common, canonical HOFs. We picked `take-while` for its similarlity to `filter`. (With state, `take-while` can be expressed using `filter`, but we are operating in a purely functional setting.) We chose `ormap` because it produces non-list output and because students had seen it in the text but had not had to use it, unlike its sibling `andmap`, requiring some transfer.

Prior to our study, students had seen all these functions except `take-while`. However, they had not been exposed to a *systematic* way of thinking about the behavioral features of these functions, as given by the columns in the figure. These functions have slightly different names and argument orders in individual languages. In our study we allowed students to write free-form text that we interpreted, allowing students to use any reasonable names for these functions.

5 PEDAGOGIC CONTEXT

This work is situated in a highly competitive private university (post-secondary) in the USA. The setting is an accelerated introductory course. About two-thirds were first-year students (typically 18 years old); the rest had already had at least a semester of post-secondary study. About 10% had no prior computing, with the rest having taken some (high school) computer science, as much as the AP CS A course, with a handful having gone farther. Nearly all were new to functional programming.

To place into the accelerated course, all students had to complete a month-long module (over Summer 2020) that teaches beginning functional programming using *How to Design Programs* [8], with assessment approximately every ten days. That material roughly compares to the first month of a conventionally-paced introductory course at the same university. By the end of the module, students were introduced to HOFs in Part Three (https://htdp.org/2019-02-24/part_three.html) of the book, which teaches them as abstractions over code. The fourth and last assessment asked students to redo the *same* problems as in the third assessment, but using *only* higher-order functions and *no* explicit recursion. This assignment required students to work with the functions `map`, `filter`, `fold` (as `foldr`, not `foldl`), and `andmap` (the latter is analogous to `ormap` as shown in the table), often in composition. There were 5 problems total, including stripping the vowels out of a list of words (`filter` and `map`), producing the subset of words whose characters were all in a reference list (`filter` and `andmap`), and produce a list of unique elements from an input list (`fold`, with or without `filter`).

The first four stages of our study took place between the end of the placement process (August 2020) and the start of the accelerated course (September 2020), after students received acceptance decisions into the course. The fifth and final stage of the study reported in this paper occurred at the end of the semester.

During the semester (Fall 2020), students were encouraged but not required to use HOFs. Though they saw HOFs used repeatedly, they were not given any further instruction on them, assuming the summer preparation (including the first four stages) were sufficient.

The placement and first two-thirds of the course proper used pure functional programming, with state introduced and used extensively in the last third of the course. The placement process was entirely in Racket, while the semester was entirely in Pyret.

6 INSTRUMENTS AND METHODS

Our studies use two different representations of examples of HOF behavior. The first is *input/output examples*, such as[1]

```
(list "red" "green" "blue")
—»
(list 3 5 4)
```

Our full list of examples appears in fig. 2. We carefully chose the examples to explore the features in the columns of fig. 1. These *same* examples were used in all stages of our study that used examples.[2]

In the above example, a list of three strings is turned into a list of three numbers. These types greatly limit the set of candidate functions. Figure 1 shows why: `filter` and `take-while` produce lists of the *same type* as they consume, so they cannot possibly have this outcome; `ormap` does not produce a list at all. In contrast, `map` is a strong candidate: it is permitted to transform the type from input to output, but in doing so must preserve the number of elements; each output element must be generated in a consistent way from the corresponding input. It is easy to see that each output number is the length of its corresponding input string. This is indeed the kind of example for `map` commonly found in books and tutorials.

The astute reader will notice that `fold` could also have generated this output. Indeed, `fold` has a *universality* property (a gentle exposition is given by Hutton [18]) that makes it able to express *all* the other operators we are studying in this paper. In other words, every input/output example could (also) be generated by `fold`. We report on how we accounted for this when presenting the results of each stage of the study in section 7.

Our ground-truth label for each example is marked in fig. 2 (these were not shown to students). Our labels reserve `fold` for examples that cannot be produced by a more specialized HOF.

The second instrument showed behavior through *diagrams*, which are shown in fig. 3. The diagrams use space and color to capture key features of most HOFs from fig. 1. (A) is intended to show a `map`; (B) a `filter`; (C) is an intentionally-ambiguous distractor (it could be interpreted as a `filter`, though the lack of line going to the end might suggest a computation stopping short, like `take-while`, except the whole prefix is not included); and (D) is (one case of) `fold`, though it can also be specialized versions of it like `ormap`. Students had *not* been introduced to these images before, nor were they formally explained.

Study Tasks. Students used these representations in two styles of tasks: the *clustering* tasks asked students to form clusters of the textual examples (fig. 2) and suggest names for the clusters. The *classification* tasks asked students to label the diagrams and the textual examples with descriptions or names of HOFs that could produce the depicted behavior.[3] Section 7 provides the motivations, instructions, and findings for five study stages using these tasks.

Methods. Data collection for each stage occurred through Google Forms with identity collection turned on for four of the five stages (we accidentally forgot to enable identity gathering for the fourth stage). Students were provided written instructions on the course discussion forum including a link to participate in the study. Students were encouraged to do all the stages (motivated as tasks that would help prepare them for the course), but due to the various pressures created by COVID-19, all were made optional. The fifth stage (also optional) was conducted right after the last task of the semester, three months after the end of the fourth study.

Critically, we two authors (one of who was the class professor; the other was not affiliated with the course) agreed on how to classify each example. Since there are only 21 examples, there were not enough items to use a process such as traditional inter-coder reliability. Instead, we simply discussed each example until we arrived at complete agreement. The examples were ordered randomly (as shown in the figure) so that individual function examples did not all group together; that same order was used in all stages. Once we agreed on the labels, one author ran the analyses and did any manual coding.

Population. A total of 114 students finished the course. About 20% were female. Only a few were Black or Hispanic/Latinx. The number of students responding to each stage is as follows: $N_1 = 68$, $N_2 = 70$, $N_3 = 76$, $N_4 = 64$, and $N_5 = 83$. The students responding are not all the same, and range over about half to two-thirds of the class. At least 28 students did all of Stages 1, 2, and 5, with at least 38 doing both Stage 1 and Stage 5 (an exact count is difficult because some students switched their email addresses between rounds and cannot reliably be mapped, so these are slight under-counts).

[1]The examples are written in the syntax of Racket [11], following *How to Design Programs* [8], but because they involve only data and no code, they are extremely straightforward to translate to any number of other programming languages.

[2]Except for two small edits made after Stage 1 to reduce confusion: example 11's input's last element changed from 1 to 5, and example 16's input went from `1 7 3 -1 -4 2 8 9 -5` to `1 7 2 3 -1 -4 2 7 8 9 -5`. Figure 2 incorporates these two edits.

[3]Our terminology is directly inspired by the corresponding terms in data science and machine learning: "clustering" puts together like objects into groups (without necessarily giving them a semantically-meaningful label), whereas "classification" applies semantic labels to objects.

```
1 FILTER/TAKE-WHILE
(list "cs019" "ma054" "cs033" "cs018" "visa039")
—»
(list "cs019")

2 MAP
(list 2 1 3)
—»
(list (list "a" "a") (list "a") (list "a" "a" "a"))

3 ANYTHING
(list 1 2 3 4)
—»
(list 1 2 3 4)

4 FILTER
(list "cs019" "ma054" "cs033" "cs018" "visa039")
—»
(list "ma054" "visa039"))

5 FILTER
(list (list "a") (list "b") (list "d") (list "e"))
—»
(list (list "a") (list "e"))

6 Anything but MAP
(list 4 6 2 1)
—»
empty

7 FOLD (but could be a MAP that takes list suffixes)
(list 1 2 3 4)
—»
(list (list 10 6 3 1) (list 6 3 1) (list 3 1) (list 1))

8 MAP
(list "red" "green" "blue")
—»
(list 3 5 4)

9 ORMAP
(list true true false true false true false)
—»
true

10 FOLD (AVERAGE)
(list 1 4 4 2 6 1)
—»
3

11 MAP
(list 4 6 2 5)
—»
(list 1 1 1 1)

12 TAKE-WHILE
(list true true false true false true false)
—»
(list true true))

13 FILTER
(list 1 4 4 2 6 1)
—»
(list 1 1)

14 FOLD
(list "cs019" "ma054" "cs033" "cs018" "visa039")
—»
2

15 FOLD
(list 1 2 3 4 5)
—»
(list 1 4 9 25)

16 TAKE-WHILE
(list 1 7 2 3 -1 -4 2 7 8 9 -5)
—»
(list 1 7 2 3)

17 FILTER/TAKE-WHILE
(list (list 2 3) (list 1) (list 4 5 2) (list) (list 2 7))
—»
(list (list 2 3) (list 1) (list 4 5 2))

18 FOLD (or MAP: lam(x): 4 - x end)
(list 1 2 3)
—»
(list 3 2 1)

19 MAP
(list add1 sub1)
—»
(list (list 2 3 4) (list 0 1 2))

20 FOLD
(list 1 2 3 4)
—»
24

21 FILTER/TAKE-WHILE
(list 1 7 3 -1 -4)
—»
(list 1 7 3)
```

Figure 2: The input/output examples used in the study, annotated with their ground-truth labels

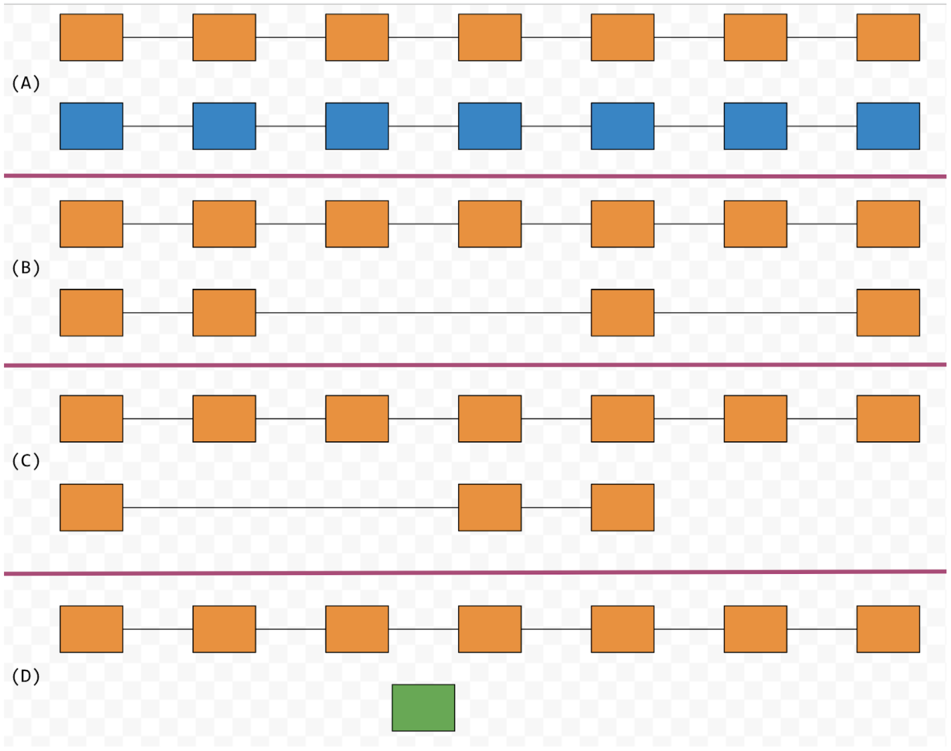

Figure 3: Images to Label (most boxes are orange, except the second row of (A) is blue and the second row of (D) is green)

7 STUDY DESCRIPTION AND ANALYSIS

7.1 Stages 1 and 2: Clustering

In Stages 1 and 2, students were given the full list of examples (fig. 2) (without the HOF labels) and asked to construct *clusters*. The specific instructions were as follows:

> Below we give you a list of 21 input/output pairs. Your task is to *cluster* them into the ones you feel are similar.
>
> What is "clustering"? That means making groups of things where all the elements in one group are similar to each other, but different from the elements of the other groups.
>
> When should two pairs go in the same cluster? When the two pairs could have been processed by the same higher-order function (possibly including ones you haven't seen before). Therefore, you want to think about relationships between the input and output in each pair (type, order, etc.). Within each cluster, all examples should share these relationships. We will ask you to describe these relationships after you form your clusters. Identifying these relationships is the heart of this exercise.
>
> You can make as few or as many (up to 8) clusters as you like.
>
> You may find pairs that could fit in multiple clusters. If that happens, make one separate cluster for all of them, and explain why you think they are ambiguous.

Students were also asked to give meaningful names to the clusters, and were specifically told, "Note that a good name for a cluster might be a higher-order function that would perform this mapping (when supplied with a suitable function as a parameter)."

7.1.1 Analysis. We created ground-truth clusters based on the labels on the examples in fig. 2. This yields six clusters (`map`, `filter`, `fold`, `ormap`, `takewhile`, `anything`). We reserved `fold` for examples that cannot be produced by a more specialized HOF. We compared each student's clusters to our ground-truth ones. Since these are set comparisons, we built atop the Jaccard index,[4] which gives a score between 0 (nothing in common) and 1 (sets are identical). Our exact scoring function is shown in fig. 4.

With 6 ground-truth clusters, a perfect score would be 1 per cluster, for a total score of 6. The lowest score is not actually 0: because every example is forced to end up in *some* cluster, every student cluster will match *some* ground-truth cluster. Indeed, our lowest observed score was 0.33, for a student who put *all* examples into a single cluster (noting that all could be implemented through the universal `fold`: section 6).

Observe that our scoring function can produce a generous overcount! For a given student, several of their clusters could be close to one ground-truth cluster. Each of these would get a fairly high score. Instead, for each student cluster, we should remove the ground-truth cluster that yields the highest score, and proceed only with the remaining clusters. (Technically, this greedy algorithm may not

[4]The Jaccard index (or Jaccard similarity) of two sets, A and B, is $|A \cap B|/|A \cup B|$.

```
For each student:
    For each student cluster:
        For each ground-truth cluster:
            Computed the Jaccard index between that student and ground-truth cluster.
        Assign the student cluster its highest Jaccard index across all ground-truth clusters.
    Sum the scores over all student clusters.
```

Figure 4: Our algorithm for scoring clusters

yield the highest possible score; instead, we have to consider all such pairings of clusters and find the maximal summed score.) We chose not to adjust for this overscoring in our analysis.

7.1.2 Findings. After Stage 1, we found a mean $\mu = 2.51$ (with $\sigma^2 = 0.66, \sigma = 0.81$) and median of 2.52. These outcomes suggest that students have some ability to perform this classification, even early into their use of HOFs. In particular, many students were able to recognize basic `map`- and `filter`-like behavior. However, several things stood out from their written responses:

- A failure to appreciate that multiple operators can produce the same results. We believe there is a certain degree of behavior akin to *functional fixedness* [7] in these answers: if one operator produces a certain outcome, another one will not.
- Difficulty with `fold`, which is anyway one of the more complex HOFs (since it simulates an accumulating loop).
- Many cases where they were unable to use an operator *name* even if they provided a meaningful *description*: e.g., “Member-wise, order-retaining function application” in place of “`map`”.
- Several answers focused on the *types* but not on the *operations*: e.g., “Non-nested List of Integers” or “returns a list of lists” as the label of a cluster.
- Looking at the order in which students presented their clusters, earlier ones tended to have much crisper definitions, and much more often corresponded to names of operators. Later clusters were much more narrow and even explicitly included guesses (e.g., “Reverse?”).

In the context of our research questions, we thus see many students recognizing features (such as order, output length, and return type) and connecting some HOF behavior to HOF names. Students struggled more to describe their clusters when they could not connect them to HOF names. Overall, the gaps between our ground truth and their clusterings suggested that several students were still well short of mastery of how the examples aligned with specific HOFs.

After Stage 1, we noticed two problems with our examples that may have caused students to focus on overly-specific properties (see footnote in section 6). After making slight adjustments to avoid them, and improving the clustering instructions to avoid some confusions we noticed, we asked students to try again (with no feedback or additional directions). Note that both these factors, combined with familiarity with the examples, could themselves cause students to do a little better. Using the same scoring scheme, this time student results produced a mean $\mu = 2.64$ (with $\sigma^2 = 0.70, \sigma = 0.83$) and median of 2.51. This appears similar to the results from Stage 1. We found 43 students had submitted responses to both stages, so we performed a within-students comparison using a paired two-sample for means t-test. For a significance level of $\alpha = 0.05$, we obtained a p-value of 0.21, indicating no significant change for these students.

Nevertheless, we saw notable improvements in the prose, if for no other reason perhaps because of practice and because our instructions had improved. We still saw several of the phenomena described above (e.g., “Keep Specific Elements” as a label in place of “`filter`”). Given the persistence of these traits, we decided to try an alternate strategy.

7.2 Stage 3: Labeling

Following the two clustering exercises, we explained the intent of the exercises to the students and provided (a version of) the chart in fig. 1. We also decided that the classification task was both too onerous and too ambiguous (due to the lack of strict partitioning of elements—both because some examples were intentionally ambiguous, and because of the universality of `fold`). We decided to try again with a lighter-weight (and potentially less ambiguous) activity.

The new activity presented the diagrams from fig. 3 and asked students to *label* them. The instructions read:

> For the four pictures below, give each as meaningful and concise a label as possible, and tell us why you picked it. It may be helpful to think in terms of the characteristics you may have used in the previous quiz.

Along with these directions, the instructor told the students, “After you’re done I’ll post my thoughts on these exercises, so you’ll have it fresh in your head in time for the start of class.” (Recall that these exercises occurred before the start of the semester, but after students finished the placement process to be admitted to the course.)

7.2.1 Analysis. Because each picture had been drawn with specific a function label in mind (except in the case of the distractor), we could simply compare student labels against our intent. One author performed the evaluation.

7.2.2 Findings. Since the diagrams directly highlighted certain HOF features—notably element type, output type, and output length—we hoped students would have an easier time associating the diagrams with specific HOFs. Our findings were somewhat surprising to us.[5]

For (A), `map` (the intended HOF) was indeed the most common answer (just over half used the word “map” in some reasonable form, mostly just “map” itself). A handful used what are effectively

[5]In retrospect, we realize that our figures were probably too abstract. As one possibility to consider, Rodrigo Duran has suggested the use of shapes in addition to colors, to reduce the abstraction leap.

synonyms, like "transform-all", "Element-wise function application", or "go-through-all single-input-dependent". Some were less crisp, e.g., "modifies every element" or "All elements processed to obtain products in list". Some were ambiguous, mixing function and visual descriptions: "Orange rectangles mapped one-to-one to blue" or "Map Orange "list" to Blue "list"". Finally, several were fixated purely on the visual: e.g., "change-color", "Color swap", or "Convert all to blue". 10% focused purely on colors.

(B) was intended to be `filter`, and the responses fit the same pattern:

- Over half used the word "filter" explicitly in some form.
- A handful used synonyms, like "Select".
- Some used phrases that are vaguely reminiscent but insufficiently descriptive: "Returning subset of elements in list of original length", "Replace", "Search and Return".
- About 10% focused on physical attributes: e.g., "Remove boxes 3, 4, and 6", "Groups of Boxes have the same total length of connected boxes but Number of Boxes are different for each group", "unchain".
- A few combined abstract and physical descriptions: "Selecting some / filtering 7 orange rectangles", "Filter Orange "list" to "list" of oranges".

Recall that (C) was a distractor. The responses, however, suggest that students picked up on the visual quite well:

- About a third of students labeled it as some version of `filter`.
- About one fifth interpreted the short line as truncation, made up a description of an abbreviating `filter`: e.g., "Filter with Exit", "filter-until", "Select and truncate", "filter-like", "Filter & Cut", "Slice and filter", "Limited Filter?", "partial-filter".
- Perhaps surprisingly, very few students made physical interpretations: "Selecting some / filtering 7 orange rectangles", "filter_five", "unchain-and-shorten".
- About 10% of answers were difficult to understand and classify (e.g., "filter 2x", "Select firsts", "Different Number of Boxes and Total Box Group Length"), perhaps reflecting the distractor nature of the problem.

Finally, (D) was intended to represent a `fold` or special cases of it that don't produce lists, like `ormap`. Half the students chose accurate names like `fold`, `reduce`, and `ormap`. However, numerous students were thrown off by the single value of a different color, and focused on that characteristic: there were numerous responses like "returns new value", "manipulateAllBoxesToOne", "collapse". Some were unable to express any useful high-level characteristic, such as "generates a result based on inputs" (which describes *all* functions!). A few were arguably incorrect, like "Constant, different type" or "count-filtered". Finally, we again saw about 8% of students combine form and function: e.g., "Foldl/r Orange 'list' to Green", "Folding 7 rectangles to any color", "Reduce and change-color", "Condense to one green box".

Overall, we were surprised to see over 10% of students focusing at least partly on the visual notation (including color) rather than the depicted behavior. This suggests that students hadn't really grasped the point of the activity, despite having that described as part of the post that released the exercise.

We did notice that many of the students who focused *purely* on the visual had not completed the previous stages, so they may not have fully grasped the task, and may not have paid attention to the instructions. In contrast, most of the students who gave hybrid answers had completed one or both prior stages. Nevertheless, our general take-away was that the diagram-based version was perhaps too open to interpretation. In light of this, we did not do in-depth comparisons of results for students who completed both Stage 3 and one of the first two stages (at least 56 students).

Relative to our research questions, we find roughly half of the students are accurately associating patterns with HOF names (RQ 2). The others are detecting some patterns, but they aren't all behavioral and they aren't necessarily aligned with the functions that students learned during the initial pedagogy (RQ 1). Based on the responses that emphasized the visual features of the diagrams, we decided to try the labeling-oriented again, but this time on the textual examples.

7.3 Stages 4 and 5: Classifying

Contrary to an earlier promise that Stage 3 would be the last one, the instructor posted one "last step", accompanied by a lengthy explanation. (Space precludes including the entire text.) It briefly mentioned the educational goal, then described how the instructor thinks about functions in terms of their characteristics. The post shared the table shown in fig. 1. Finally, students were told, "Based on all this, I would like to request you to fill out one *truly* final quiz, which should wrap up your training on this topic." The specific instructions read:

> For each pair, please select *all* the higher-order functions that could (reasonably) have transformed the input into the corresponding output.

Students were asked to check one or more boxes labeled `map`, `filter`, `take-while`, `fold`, and "Other" (which let them enter free-form text). We intentionally left out `ormap`, so students would have to enter it using "Other".

7.3.1 Analysis. We compared student labels to our ground-truth labels (from fig. 2). For all examples, we accepted `fold` responses: e.g., a response of `filter` and `fold` was put in the same bin as one of `filter` alone. Thus students were never penalized for using the more general operation in addition to the more specific one. In the discussion that follows, we will discuss `fold` more specifically for particular examples, and as a general phenomenon.

7.3.2 Findings Before The Course Started (Stage 4). The left-hand chart in fig. 5 displays the frequency of errors (y-axis) as a % of responses, binned by every 5% (x-axis). To make clear how to read the graph: it says that 6 (y-axis) of the 21 examples have fewer than 5% of responses erroneous, but one example has 70-75% (specifically, 73.4%) responses wrong. Because the examples are all so different, we do not believe summary statistics about these errors would be meaningful.

Using the notation E*n* to mean the example *n* from fig. 2, the low error rates are on `map` (E2(1.6%), E8(0%), E11(3.1%), E19(12.5%)), `fold` (E10(0%), E14(4.7%), E20(0%)), and `map`/`fold` (E18(12.5%)). We put them in example- rather than percentage-order to make them easier to look up in fig. 2.

The errors are somewhat higher for the `filter`-only examples (E4(15.6%), E5(17.2%), E13(14.1%)) for a simple reason: students

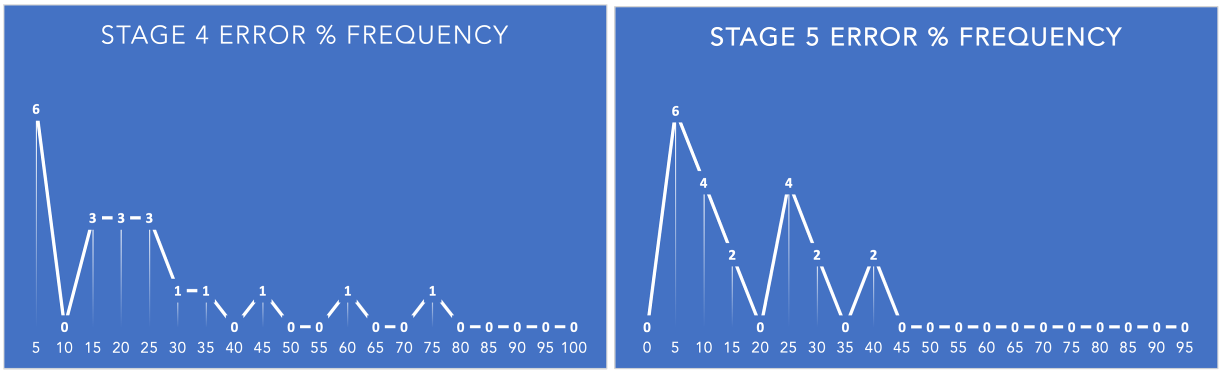

Figure 5: Error Frequencies in Stages 4 (left) and 5 (right)

routinely also tag these as fitting `take-while`! *All* the errors on those examples are caused by this confusion.

On the two examples that were `take-while`-only, the error rates were somewhat higher: E12(25.0%) and E16(26.6%). All the errors come from students incorrectly also marking `filter`. In one instance, on E12, a student marked *only* fold.[6] The overall numbers suggest a potential bias where students are more likely to misclassify `take-while` as `filter` than vice versa. We do not have an explanation for this phenomenon (if true), though one conjecture is their greater familiarity at that point with `filter`.

On two examples that could be either `filter` or `take-while`, error rates were not too high: E1(18.8%), E21(17.2%). In both cases, the only errors were when students forgot one or the other possibility. However, the bias is not uniform. In the former (E1), 14.2% of students (not of the errors!) neglected the `take-while` possibility.[7] In the latter (E21), 15.9% overlooked the `filter` possibility. On the third such example, the error rate is nearly double: E17(34.4%). Here, 3% overlooked `take-while` but ten times as many overlooked `filter`, perhaps because the empty list looks like a sentinel. At any rate, it is notable that across all these examples, students never confuse either for `map`, only for each other. This suggests that students have accurately observed the HOF feature regarding whether all items or a subset are in the output, but are perhaps not attending to *which* elements are being dropped (a suffix as in `take-while` versus interspersed as in `filter`).

Three high-level mistakes arise in the five high-error examples.

- *Failure to mark all appropriate* HOFs: On both E3(56.3%) and E6(23.4%), multiple HOFs beyond `fold` applied. On E3, 60% of errors came from omitting just `fold`, while the other 40% omitted another applicable function. On E6, all options other than `map` could have applied, and indeed nobody selected it (the omissions were among the other options). This again suggests a robust understanding of the HOF feature regarding whether the output lists and the input lists have the same length, but not a good understanding of `fold`.
- *Marking* HOFs *whose types fail to match the examples*: On E9(23.4%), most of the erroneous answers chose *only* `filter` and/or `take-while`, neither of which is even type-correct (because a single boolean, not a list, was returned). Some chose one or both of these in addition to `ormap`, which is type-correct. This suggests that students were attending more to the relative length of the input rather than the type.
- *Mishandling details later in the lists*: E15(40.6%) was intentionally subtle, and chosen as an attention test: at quick glance, it looks like a `map` (of a squaring function), but the input has five elements whereas the output has only four. Many students used the "other" option to write in a statement, e.g., wondering if the problem was incorrect, or indicating that had a `16` been present, they would classify it a `map` (we classified these as correct).

Throughout our data, we see evidence that students struggle with `fold` (the general-purpose HOF). As we have noted, `fold` is the hardest of the HOFs to use, presumably because it supports all of the core features (transforming, skipping, and aggregating elements). Indeed, one way to understand `fold` is through this multi-feature support (though this is a hard perspective to detect based on its unabstracted code or types). The data for E15(40.6%) showed that students had not fully grasped this perspective: 13.3% of students chose both `map` and one of `filter` or `take-while`, choices which recognize that the example features both transformation and skipped elements. Ideally, students would understand that such situations call for `fold` (or a composition of HOFs, which 23% of students wrote in).

E7(73.4%) is a slightly tricky example that also raises questions about which features students might be perceiving (even the authors originally mis-classified it, intending it as a `fold` problem but failing to notice that it can also be a `map`). The example preserves list length (suggesting `map`), but the transformation of elements is not immediately obvious (suggesting `fold`). One student suggested (free-form) a list-building function, `build-list`, which is loosely applicable inasmuch as it is often used to generate sequences of

[6] It is possible that the idea of using `filter` and `take-while` on lists of Booleans (which is quite rarely done in practice, and had not been done on the work they had seen up to that point) was sufficiently unfamiliar that they failed to recognize these as possibilities.
[7] In retrospect, this example was poorly designed: because the course's name is "cs19", students may have instinctively expected one is filtering for the present course. We did not spot this until writing this paper.

Example	Labels	Stage 4 Error Rate	Stage 5 Error Rate	Improvement
1	`filter,take-while`	18.8	22.9	-4.1
2	`map`	1.6	1.2	0.4
3	anything	56.3	14.5	41.8
4	`filter`	15.6	8.4	7.2
5	`filter`	17.2	10.8	6.3
6	anything but map	23.4	20.5	3.0
7	`fold`	73.4	36.1	37.3
8	`map`	0.0	1.2	-1.2
9	`ormap`	23.4	21.7	1.8
10	`fold`	0.0	2.4	-2.4
11	`map`	3.1	1.2	1.9
12	`take-while`	25.0	24.1	0.9
13	`filter`	14.1	6.0	8.0
14	`fold`	4.7	9.6	-5.0
15	`fold`	40.6	28.9	11.7
16	`take-while`	26.6	27.7	-1.1
17	`filter,take-while`	34.4	39.8	-5.4
18	`fold,map`	12.5	6.0	6.5
19	`map`	12.5	3.6	8.9
20	`fold`	0.0	1.2	-1.2
21	`filter,take-while`	17.2	21.7	-4.5

Figure 6: Error Rates and Difference

numbers, but cannot work here because it would lead to a type error (it consumes only *one* number, in addition to a function).

Overall, these results support a conjecture that students are generally learning to identify key features of input/output examples and which HOFs are associated with each of those features. Where students make errors, they have not pushed far enough on those associations, failing to notice inconsistencies in types or details in how individual HOFs could be composed in actual code.

7.3.3 Findings Near the End of the Course (Stage 5). We might expect that students' concepts of HOFs would evolve after using them throughout the course after the first four stages. Accordingly, at the end of the semester (three months after Stage 4), we repeated the Stage 4 study. Figure 5 contrasts the error frequencies for the two stages. Figure 6 shows the changes in the class-wide error rates on each example between the fourth and fifth stages.

Both the figure and the charts show some significant gains alongside some worsening performance. That said, *we cannot draw too much knowledge (or comfort) from these results.* Unfortunately, we are unable to provide a per-student comparison as we critically failed to record student identities in the Stage 4 data. However, we do know that the overlap between Stages 3 and 5 is at least 51 students, so we can hope for about 40 students in common between 4 and 5. Nevertheless, irrespective of overlap size, it should hardly be surprising if a semester of use (and teaching) of and with this concept should generally improve student performance.

More troubling, perhaps, is that some of the errors seen earlier persist. On E1, 16% of students missed `take-while`. On E21, 18% missed `filter`. On E17, 36.1% missed `filter`. These error rates are actually higher than on Stage 4! We caution that these may be due to differences in population; also, the examples may have been fresher in the minds of students at Stage 4. Still, coming at the end of a semester of frequent use, these numbers are telling. Similar patterns persist across other examples.[8] These similarities suggest either that our populations are similar or even perhaps that these are widespread mistakes even amongst programmers with some experience.

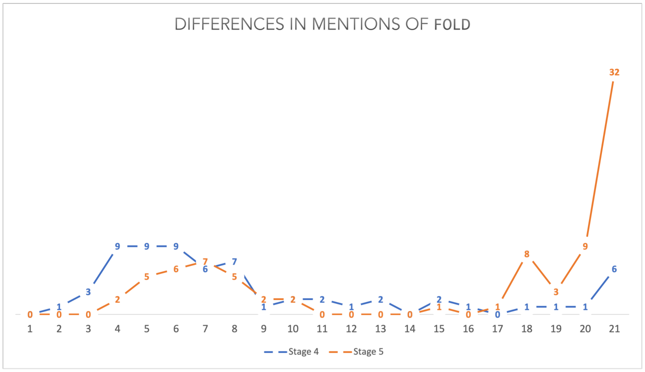

Figure 7: Frequency of `fold` in Stages 4 and 5

The Frequency of Fold. Our discussion of Stage 4 pointed to challenges that students appeared to have in using `fold`. As students gained more facility with HOFs during the semester, we might see

[8]In addition, three students chose almost all functions for all examples. It is unclear whether these were chosen with intent or by students who did not feel like doing the exercise. We include these in our error rates, but they account for most of the small declines, and mask some of the improvements.

them recognize more problems as being instances of `fold`. Do we see that in our data?

Indeed we do. In fig. 7 we graph the frequencies with which students mention `fold` in their classification in Stages 4 and 5. There appears to be a significantly greater number of students who label most or all problems with `fold`. The summary statistics confirm this: for Stage 4, a mean of 8.79 and median of 7; for Stage 5, a mean of 15.43 and median of 19. Because of the potential differences in population we do not try to interpret these differences further, but simply record their presence and suggest that they are not inconsistent with the growth we would expect to see in students.

7.4 Threats to Validity

There are natural threats to generalizability owing to population sizes, locations, class-specifics, choices of HOFs, and the effects of COVID-19. We list below more interesting considerations.

Threats to Internal Validity. Different subsets of students participated in each stage, which limits the conclusions we can draw from our data. Out of the 114 students in the course, a low of 64 and a high of 83 participated in the individual stages (the population data in section 6 suggests roughly 35 students participated in all stages) . Once students knew the instructional aims of the study, stronger students may have stopped participating, which could have skewed the data. In addition, classifying could be an easier task than clustering, which could explain why students appear to do better at the end. These said, the persistence of certain errors into Stage 5 indicates that stronger methods for teaching about HOFs are still in order.

Ecological Validity. Ultimately, we care about students applying HOFs correctly while programming. We do not know whether students make programming errors consistent with the confusions that our data show. For instance, do students accidentally use `take-while` where they should use `filter` or vice versa? It is difficult to tell because it is possible students do make this mistake but correct it before they submit their work—so the cost may not be in eventual correctness but rather in frustration and time.

A related concern is that we never ask students to explicitly write the functional parameter. While this does not affect our research questions, it does impact students' ability to transition from a behavioral understanding to actually using these HOFs. This can especially apply in situations like `fold`, where by semester's end many students may "know" these are all instances of `fold`, but some of them may not necessarily be able to actually use `fold` to produce the described behavior.

8 DISCUSSION

Research Questions. Our research questions asked whether students were attuned to the feature-space of HOFs and able to connect input/output examples to specific HOFs. We find few basic errors: students don't confuse `map` and `filter`, which differ on multiple features including the relationship between the input list and the output list. They are more likely to confuse `filter` and `take-while`, which differ only on which elements are retained in the output list. It is worth noting that not all developers can make this distinction either: as just one example, an active StackOverflow user [22] who is a self-declared "Java Developer" with a high reputation score (as of 2021-03-14) notices [23] the similarity in type and, on a particular example, behavior between these two functions and asks "What was the need of this new function then?" One can see several questions along these lines on the Web [27]. (To be fair, these developers may be used to thinking about the imperative simulation of `take-while` using `filter`.)

Overall, our data suggest that a large fraction of participating students are able to do fairly well on HOFs, initially with only a little training and assessment, followed by a few iterations of our instruments. In particular, though they may have benefited from familiarity through repetition, it is critical to note that they were *not* given any individual assessment that would guide their correction (excluding Stage 5, by which time they would have received feedback in the context of the course). Any correction presumably came from their own observation.

We do still see several students making labeling errors that are not type-consistent, and issues with students not knowing the applicability of `fold`. One key question is the extent to which these issues cause problems while students are actually programming. Type errors would be flagged at compile- or run-time. Failing to identify a potential use of `fold` might not lead to incorrect code, just longer code. Our overall project has yet to address how the issues we detect through our study instruments manifest in programming behavior, productivity, or experience. That would be best done with a separate component that watched what students do while coding.

Students' free-form descriptions of clusters in the first two stages show that many students struggle to provide crisp descriptions of HOF behavior. This is hardly a new observation: students, as well as programmers, can struggle to write precise behavioral specifications [2, 17]. Whether direct instruction on describing HOFs via features would help students develop this skill is an open question.

Reflection on our Study Design. Ours is the first study we have seen on the development of understanding of HOFs. As such, lessons regarding our methodology are themselves a contribution of this paper. Knowing what we do now, what changes would we make to our instruments and methods?

- *We would start with the classification task rather than the clustering task.* The classification task focuses on which features of HOFs students are recognizing. The clustering task, in contrast, requires students to both perceive features and decide how to group them. This conflation makes it harder for us to tease apart which conceptions led to clustering decisions. (In addition, Barsalou [3] points to additional considerations for the design of categorization tasks.)
- *We would ask for justifications of classifications on some examples.* Our current design, which only asked students to label examples, doesn't tell us which features students saw in an example (especially one in which multiple features arise). We have inferred the features from the HOFs a student selected, but students might not be actively thinking through the features when making these selections.
- *We would include an activity that has students write illustrative examples of the HOFs for themselves.* Recognizing features within a given input/output example is a different activity

than being able to generate input/output examples that reflect the defining characteristics that distinguish HOF. We are curious as to which of the HOF features students would touch upon if asked to generate the examples for themselves.

We are also rethinking our use of the diagrams from Stage 3, given that they did not seem to improve student performance much. The diagrams explicitly summarize the types and length features of the HOFs that we included in this study. That elements may be transformed is implicit in the types, but students might not recognize that feature without additional indicators, such as shapes, in the diagrams. From a methodological standpoint, it would be useful to know which features students perceive in the diagrams so that we could pick an appropriate role for these diagrams in both study instruments and pedagogy surrounding HOFs.

Static Types. Before Stages 1 through 4, students were exposed to HOFs through the dynamically typed version of Racket. Following *How to Design Programs*, students saw type specifications as comments, but they were not checked. Therefore, students did not have the benefit of static types. During the semester, they used Pyret, a language with an optional static type checker, that they were at least briefly required to use. It is unclear how our results would change if students worked in a typed language all along, especially given the mathematically interesting relationships between types and the set of allowable program behaviors [28].

We see relatively little confusion when the types of functions are sufficiently distinct: e.g., students rarely confuse `map` with `filter` or `take-while` (but routinely confuse the latter two). However, this phenomenon may not generalize: in our study students worked with a very small number of HOFs, so they could have easily just enumerated all the ones they knew. In real languages there can be dozens of such functions, several with overlapping types and thus similar but distinct functionality. Our study suggests that, at least for novices, much more effort needs to be put into distinguishing those functions that are similar in type.

From Categorization to Misconceptions. The categorization processes we have used tell us only so much. They can point to mass trends, and in particular can identify common weaknesses. But without more narrative from students, they cannot tell us *why* they made those choices.

Given that this was a first study in this space, we were unsure of how much effort it would take students, and owing to COVID-19, wanted to keep the stages especially low in burden. A future study should, however, do more to ask students about the reasoning behind their choices. We expect this will tap a rich vein of information, even in cases where they chose the "right" answer (but for "wrong" reasons). We believe that as a research community, we do not understand student conceptions and misconceptions of HOFs very well at all. It would also be interesting to relate findings at this level to those on a larger scale, such as Khatchadourian, et al.'s analysis of Java streams [19].

Implications for Planning. This paper was partially motivated by the value of planning in contemporary styles of programming, especially data-centric code. Some prior planning work [9, 10] shows success when students are comfortable with HOFs, which in turn Wickham shows [29] connect well to standard data manipulation. However, none of this work has focused on what it takes to make students comfortable with and develop a good "feel" for these operations. Our paper suggests techniques for helping them understand these behaviorally.

Beyond correcting for the flaws described above, there are (at least) two parts still missing from this work on the path to making students better at planning:

- HOFs, by definition, are incomplete behavioral specifications: a great deal depends on the function parameter passed to them. Our work has ignored this parameter, asking students to imagine which ones could *possibly* be passed. Focusing on the function parameter—e.g., asking students to fill in the actual function that *would* map a given input to the given output—should yield new insights into student understanding of HOFs. We conjecture that some students can correctly identify which HOF can perform a transformation but might have difficulty filling in the actual functional parameter; would students also have difficulty going in the opposite direction?
- This paper has pointedly focused on one function at a time, ignoring function *composition.* Once we start to consider compositions, the space of possible answers for our clustering and classifying tasks becomes much richer (and thus perhaps much harder to manage). However, function composition is critical for achieving the goal of planning. Thus, we need methods that can tame this complexity and enable us to help students develop behavioral conceptions of the compositions of functions (higher-order or otherwise).

ACKNOWLEDGMENTS

This work is partially supported by the US National Science Foundation. We are especially grateful to the students of CSCI 0190, who cheerfully worked through these exercises. We thank Rob Goldstone for instructive conversations. Elijah Rivera and our reviewers provided insightful comments which helped us clarify our presentation.

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman. 1985. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA.
[2] V. L. Almstrum. 1996. Investigating student difficulties with mathematical logic. In *Teaching and Learning Formal Methods.* Academic Press, 131–160.
[3] Lawrence W. Barsalou. 1994. Deriving Categories to Achieve Goals. *Psychology of Learning and Motivation* 27 (1994).
[4] Michael J. Clancy and Marcia C. Linn. 1999. Patterns and Pedagogy. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (New Orleans, Louisiana, USA) *(SIGCSE '99).* Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/299649.299673
[5] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04).* USENIX Association, USA, 10.
[6] Ekwa Duala-Ekoko and Martin P. Robillard. 2012. Asking and answering questions about unfamiliar APIs: an exploratory study. In *Proceedings of the International Conference on Software Engineering.* 266–276.
[7] K. Duncker. 1945. On problem solving. *Psychological Monographs* 58, 5 (1945).
[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs.* MIT Press. http://www.htdp.org/
[9] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of ICER.*
[10] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *ACM Technical Symposium on Computer Science Education.*

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

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA.

[13] D. Gentner and K. Kurtz. 2005. Relational categories. In *Categorization inside and outside the lab.* APA, Washington, DC, 151–175.

[14] Eleanor J. Gibson. 1969. *Principles of Perceptual Learning and Development.* Appleton-Century-Crofts.

[15] James J. Gibson and Eleanor J. Gibson. 1955. Perceptual Learning: Differentiation or Enrichment? *Psychological Review* 62, 1 (1955), 32–41.

[16] Andy Gordon and Simon Peyton Jones. 2021. LAMBDA: The ultimate Excel worksheet function. https://www.microsoft.com/en-us/research/blog/lambda-the-ultimatae-excel-worksheet-function/.

[17] Geoffrey L. Herman, Michael C. Loui, Lisa Kaczmarczyk, and Craig Zilles. 2012. Describing the What and Why of Students' Difficulties in Boolean Logic. *ACM Transactions on Computing Education* 12, 1, Article 3 (March 2012), 28 pages. https://doi.org/10.1145/2133797.2133800

[18] Graham Hutton. 1999. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* 9, 4 (July 1999), 355–372.

[19] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. 2020. An Empirical Study on the Use and Misuse of Java 8 Streams. In *Fundamental Approaches to Software Engineering.*

[20] K. J. Kurtz and G. Honke. 2020. Sorting out the problem of inert knowledge: Category construction to promote spontaneous transfer. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 46, 5 (2020), 803––821.

[21] Peter L. Pirolli. 1985. *Problem Solving by Analogy and Skill Acquisition in the Domain of Programming.* Ph.D. Dissertation. Carnegie Mellon University, Department of Cognitive Psychology.

[22] StackOverflow Post. [n.d.]. https://stackoverflow.com/users/706317/zhekakozlov.

[23] StackOverflow Post. [n.d.]. https://stackoverflow.com/questions/46850689/how-is-takewhile-different-from-filter.

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

[25] Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. 2021. A Theory of Robust API Knowledge. *ACM Transactions on Computing Education* 12, 1 (January 2021).

[26] Simon Thompson. 1999. *Haskell: The Craft of Functional Programming* (2 ed.). Addison-Wesley.

[27] MSDN Forum Thread. [n.d.]. https://social.msdn.microsoft.com/Forums/en-US/393da5e4-f33b-4f0d-bfc1-ae73d2cd77df/what-is-the-difference-between-takewhile-and-where-in-linq.

[28] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89).* Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

[29] Hadley Wickham. 2014. *Advanced R.* Chapman and Hall/CRC.

[30] Hadley Wickham. 2019. The Joy of Functional Programming (for Data Science). https://www.youtube.com/watch?v=bzUmK0Y07ck.