JFP **31**, e23, 17 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.
doi:10.1017/S0956796821000216

EDUCATION MATTERS

Segments: An alternative rainfall problem

PETER ACHTEN
Institute for Computing and Information Sciences, Radboud University, Nijmegen, the Netherlands
e-mail: *P.Achten@cs.ru.nl*

Abstract

Elliot Soloway's *Rainfall* problem is a well-known and well-studied problem to investigate the problem-solving strategies of programmers. Kathi Fisler investigated this programming challenge from the point of view of functional programmers. She showed that this particular challenge gives rise to five different high-level solution strategies, of which three are predominant and cover over 80% of all chosen solutions. In this study, we put forward the *Segments* problem as an alternative challenge to investigate the problem-solving skills of functional programmers. Analysis of the student solutions, their high-level solution strategies, and corresponding archetype solutions shows that the *Segments* problem gives rise to seven different high-level solution strategies that can be further divided into 17 subclasses. The *Segments* problem is particularly suited to investigate problem-solving skills that involve list processing and higher-order functions.

1 Introduction

Elliot Soloway's *Rainfall* problem is a well-studied problem designed to investigate the problem-solving strategies of programmers. It is phrased concisely as:

"Write a program that will read in integers and output their average. Stop reading when the value 99999 is input." (Elliot Soloway (1986))

The problem has been studied mostly in the context of imperative style programming languages, in which students have early access to I/O facilities, and have limited exposure to data structures, most often arrays. The situation for students in functional programming is quite different: I/O is not taught early, but recursive data structures, most often lists, and higher-order functions are. For this reason, Kathi Fisler asked the question: *"what do students from functional-first CS1 courses do with Rainfall?"* (Kathi Fisler (2014)). To investigate this, she recast existing variations to suit the context of functional programming, which we call the *Functional Rainfall* problem. It is phrased as:

"Design a program called rainfall that consumes a list of numbers representing daily rainfall amounts as entered by a user. The list may contain the number -999 indicating the

end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list."
(Kathi Fisler (2014))

The I/O component has been eliminated and is replaced with a list of numbers. Students need to direct their problem-solving skill to dealing with the sentinel value or end of input list, as well as deciding how to process non-negative list elements. Kathi Fisler (2014) proceeds to analyze 207 student submissions:

- five high-level structures are identified to classify the student solutions: *Clean First* (42%), *Clean Multiple* (21%), *Single Loop* (19%), *No Cleaning* (3%), and *Clean After* (1%). An additional category, *Unclear* (14%), covers unfeasible solutions.
- analysis of low-level errors reveals low error rates, in particular in comparison to imperative style solutions of the original *Rainfall* problem: for *Single Loop* solutions, 74% have zero errors and 21% one error; for *Clean First* solutions, 66% have zero errors and 22% have one error; for *Clean Multiple* solutions, 46% have zero errors and 23% have one error (and 2–6 errors in the remaining cases).

The analysis shows that the majority of students choose to solve the *Functional Rainfall* problem with one of the three high-level structures *Clean First*, *Clean Multiple*, and *Single Loop*, covering 82% of all solutions. The high-level structures reveal that the students need to make two mutually dependent key design decisions: how to process the input list and what to do with that result. In *Clean First*, the input list gets filtered until the sentinel value, allowing the students to use standard library functions, *sum* and *length*, to compute the average. In contrast, in the other two solutions, the students aim for obtaining the number of relevant input and sum of their values separately to compute the average. In *Clean Multiple* this is done by reducing the input list twice, to count the number of relevant inputs and the sum of their values, and in *Single Loop* the input list is reduced to the pair of these values.

Analyses as done by Kathi Fisler (2014) provide valuable information for both the instructor and students. The instructor can capture the essence of each high-level structure with an *archetype* solution. The set of archetype solutions can be used to discuss the possible different ways of solving the same problem with students. For students who are struggling with a particular concept, it can help as a model solution how to apply that concept in practice. They can be used to show the coding solutions if different design decisions are made. In the experience of the author, this is an effective way of expanding the problem-solving skill set of students.

In this study, we put forward the *Segments* problem, as a vehicle to explore the problem-solving skills of students in this way. Although it can already be used in a course after lists (and list comprehensions in particular) have been taught, in the experience of the author, it is even more suited once also higher-order functions have been taught. It is phrased as:

"Design a program called segments that consumes a list of numbers. Produce a list of all elements, without duplicates, and sorted in increasing order. Instead of containing all individual elements, these are organized as segments. A segment is either a single value x

(neither $x-1$ nor $x+1$ are in the list) or a pair of two values a and b such that a, $a+1$, ..., $b-1$, b are in the list (neither $a-1$ nor $b+1$ are in the list). The segments must be shown as strings, formatted as x for singleton segments and $a-b$ for the other segments."

Following the terminology in Kathi Fisler *et al.* (2016), the *Segments* problem has a *cleansing* component because the required result should have no duplicates and is sorted, and it has at least two *reshaping* components because numbers need to be transformed into segments, and segments need to be transformed into strings.

It is interesting to compare the *Segments* problem with the *Adding Machine* problem used in Kathi Fisler *et al.* (2016) because it also features a segment-like structure on an input list of numbers. In the *Adding Machine* problem, a segment is defined as a non-empty sequence of numbers delimited by the value zero. This is a context-independent definition, in contrast with the context-dependent version used in the *Segments* problem, so both problems require different solution strategies to handle them. Analogous to the *Functional Rainfall* problem, but slightly more complicated, the *Adding Machine* problem features a sentinel value, which is the first occurrence of at least two subsequent zeroes in the input list. In the *Segments* problem, sentinel values do not play a role.

We identify *seven* high-level solution classes that capture the different key design decisions made by the students to solve the *Segments* problem. Within these high-level solution classes, there is enough room for students to explore different paths to solve the *Segments* problem. We identify *17* subclasses and represent each with an archetype solution. We discuss how an instructor can use the archetype solutions to give feedback and guidance to students.

The *Segments* problem has been part of weekly assignments of a functional programming course that the author has been (co-)teacher of at Radboud University. (In the course, the problem was contextualized as one involving a CD collection and asked students to write a program to produce a summary of years.) We have collected and analyzed 288 student submissions from 4 different editions of the course. The first edition featured second-year computing science students and second-year artificial intelligence students, the remaining three editions were tailored specifically for artificial intelligence students (first-year students in the second edition and second-year students in the third and fourth edition).

The remainder of this paper is structured as follows. We explain the context of the functional programming course in Section 2. In Section 3, we describe the data set of student submissions. We investigate the high-level solution classes, their subclasses, and the archetype solutions in Section 4. We reflect on the merits of the approach in Section 5. We conclude in Section 6.

2 The functional programming course

The course in which we have collected the data is the introductory course to functional programming, given by the computing science department of Radboud University in study years 2015, 2016, 2018, and 2019. The 2015 edition was the last edition for second-year computing science students and second-year artificial intelligence students. It is the first

Table 1. *Functional programming course contents*

	2015	**2016**	**2018**	**2019**
1	Introduction	Introduction	Introduction	Introduction
2	Types, type classes	(recursive) types, Type classes	Type inference, (recursive) types	type inference, (recursive) types
3	Lists	List (compre-hensions)	Type classes	Type classes
4	List compre-hensions, higher-order functions	**Higher-order functions**	List (compre-hensions)	List (compre-hensions)
5	**Higher-order functions**, reduction strategies	Reduction strategies, equational reasoning	Project: Soccer-Fun	**Higher-order functions**
6	Equational reasoning	Project: Soccer-Fun	**Higher-order functions**	Reduction strategies, equational reasoning
7	(Binary search) trees	Case study: Countdown	Reduction strategies, equational reasoning	Case study: Countdown

half of a 6-EC[1] semester course (2 × 7 weeks). The 2016–2019 editions were targeted for artificial intelligence students exclusively: in 2016 for first-year students and in 2018–2019 for second-year students, which explains the year gap between 2016 and 2018. They are all a 3-EC quarter course (7 weeks). In 2015, 79 computing science students participated in the course and 66 artificial intelligence students. The number of students for 2016, 2018, and 2019 are 114, 112, and 117, respectively.

In the course, we use the pure, lazy, functional programming language *Clean* (version 2.*x*). Note that in the 2015 edition, we allowed students to also submit solutions in *Haskell* (7 out of 94 teams did this). The computing science students had been taught (the imperative core of) *C++* in the first semester, and *Java* in the second semester. The artificial intelligence students had been taught *Java* exclusively (one semester in 2016 and two semesters in the other editions).

Table 1 displays the course contents in more detail. *Project: Soccer-Fun* refers to the start of a multi-week project (Peter Achten (2011)). In 2015, it was part of the second half of the course. *Case study: Countdown* refers to explaining the Countdown solution (Graham Hutton (2002)). In all editions, the *Segments* problem was part of the problem set of the week in which higher-order functions are the key subject of the lecture, marked in boldface in Table 1. In all cases, the students have worked with lists (in 2015 and 2016

1 EC(TS) stands for European Credit Transfer and Accumulation System. One study year has a workload of 60 EC. In the Netherlands, this corresponds to 1680 hours of study, so one EC corresponds with 28 hours of study.

also the accumulator pattern) and list comprehensions and have been exposed to the standard library of list functions that is available in the host language. They are familiar with structural recursion over lists and have practiced with partitioning lists in various ways. List comprehensions provide students with access to list problem-solving techniques that would otherwise require higher-order function patterns, such as *filter*, *map*, *zipWith*, and Cartesian product. Indeed, we use list comprehensions as stepping stone to introduce these higher-order function patterns. Besides these, the subjects cover function composition, currying, lambda-abstraction, and the fold functions.

3 The data set

In all editions of the course, students work on weekly assignments. These assignments always consist of two parts. The first part consists of (small) tasks to practice with the new concepts. The second part consists of (larger) tasks designed to test the problem-solving skills of the students and thus have no prescribed way of solving the problem. In all editions, the *Segments* problem was in part two, and most of the time one of the last tasks the students worked on. We advise students to work in pairs but allow solo programming as well: we had 209 pair teams and 79 solo teams, giving a total of 288 candidate submissions of the *Segments* problem. Assignments are *formative*, and students receive feedback on their submissions from teaching assistants. We extracted the solution concerning the *Segments* problem from all submissions and anonymized them. The participation of student teams concerning the *Segments* problem was

- 36 submissions (13%) *skipped* doing the *Segments* problem. Some students added comments about running out of time or saying that they could not figure out how to solve the problem.
- 39 submissions (13%) were considered *unfeasible*, having nontrivial compiler errors or were mangled in some way (similar to the *Unclear* category in Fisler's study).
- 62 submissions (22%) limited themselves to only *only cleaning of data* (removing duplicates and sorting). Also in this case, some students added comments about running out of time or saying that they could not figure out how to solve the problem.
- 151 submissions (52%) were feasible.

We have analyzed the high-level solution strategies in the 151 feasible submissions. To see if there are dominant solution strategies, we are also interested in their relative distribution. We do not count submissions that are too identical, regardless of the cause of this similarity: in some cases, plagiarism has been detected, in others, students who did the course the second time re-used their submission of the previous year. We used the following process to determine which submissions are similar. First, the seven *Haskell* submissions were manually translated to *Clean*. Second, all comments were removed, non-essential repeating white space was reduced to a single white space character (*Clean* is layout sensitive), and all pre-amble lines (module header and import statements) were ignored. All custom function names and formal parameters were consistently renamed to

Table 2. *High-level structure classes and subclasses*

	High-level structure class	**%**		**Subclasses**	**%**
1.	Seek	21.5%	1.1	Seek & DropWhile	9.1%
			1.2	Seek & Memo	5.8%
			1.3	Seek & Filter	4.1%
			1.4	Seek & Drop	2.5%
2.	Bounds	19.8%	2.1	**Bounds Values**	**15.7%**
			2.2	Bounds Indices	4.1%
3.	Fold	16.6%	3.1	Fold	8.3%
			3.2	Fold, Actually	8.3%
4.	Count	14.9%	4.1	**Count & Drop**	**12.4%**
			4.2	Count & Continue	1.7%
			4.3	Zip Count & Drop	0.8%
5.	Accumulate	12.4%	5.1	**Accumulate**	**12.4%**
6.	Segments	11.5%	6.1	Inflate Segments	4.1%
			6.2	Deflate Segments	4.1%
			6.3	Zip Predless Success	3.3%
7.	Exotic	3.3%	7.1	Les Uns Et Les Autres	2.5%
			7.2	Zip With Diff	0.8%

id_i. On this simplified data set, the *Levenshtein distance* (Vladimir Levenshtein (1966)) was computed between all unique combinations of different submissions. We manually inspected the pairs of submissions with low Levenshtein distance value. There were 30 submissions that we considered to be identical and these were subsequently removed from the set of 151 feasible submissions, leaving us with 121 submissions for categorization.

4 The segments problem is rich in solutions

We have analyzed the 121 submissions to discover which solution strategy the student teams have adopted. This has been done in two steps: first identify the *high-level structure class* and second its *subclass*. The high-level structure class captures the *dominant idea* of the solution, and its subclasses capture the alternative ways to realize the dominant idea. This point of view is declarative rather than operational. For instance, from an operational point of view, *Fold* and *Accumulate* are very similar, but from a declarative point of view they are not: with *Fold* the student communicates that a simple value-state function gets lifted to a more complex values-state function, and with *Accumulate* the student explicitly communicates the list traversal and state management.

In this way, we have identified 7 high-level solution classes, and within these classes 17 subclasses (Table 2). The *Exotic* high-level solution class is a bit different and collects individual high-level solutions that occur sporadically. Of each high-level solution (sub)class, the relative contribution is shown (using boldface to highlight the most frequently occuring subclasses).

We first discuss the individual high-level solutions and their subclasses in Section 4.1. We analyze the code structure and code quality in Section 4.2.

4.1 The high-level solution classes

We discuss the individual high-level solution (sub)classes. We extract an *archetype* solution for all subclasses except *Les Uns Et Les Autres* (the reason for that is explained in Section 4.1.7). The purpose of an archetype solution is to highlight the key idea of a solution. In the experience of the author, working with archetype solutions is beneficial. First, using an archetype solution instead of a student submission allows the instructor and student to not get distracted by coding details that often clutter student submissions (students do receive feedback on their submission and the assignments are formative). Second, by getting to understand how their submission relates to an archetype solution, makes the student open-minded for code reviewing. Third, students see entirely different high-level ways to solve the same problem. Fourth, the instructor can use archetypes to help students make the transition between applying programming techniques. Fifth, archetypes can be used to compare solutions and explain why one is better than another.

The archetype solutions correspond to the second step of the well-structured solution. For the sake of uniformity, we use a solution that identifies boundary values, so the type of the archetype functions is *[Int] → [(Int,Int)]*. The functions have the same name as the subclass they represent.

4.1.1 Seek

In the *Seek* solutions, the key design decision is to introduce a function, *Seek*, that determines the last element b of the initial segment of an input list of numbers with head element a. Once this value has been found in the input list, the initial segment (a,b) can be produced. There are a number of ways to proceed, and these determine the subclasses:

- *Seek & Memo*: the function that seeks b memorizes a, so it can proceed recursively with the remainder of the list after b.
- *Seek & DropWhile*: proceed on the input list of numbers from which initial elements unequal to b are dropped, and drop b as well.
- *Seek & Filter*: proceed on the input list of numbers from which all elements are filtered that exceed b or are not member of the segment $[a..b]$.
- *Seek & Drop*: proceed on the input list of numbers from which the first $b-a+1$ elements are dropped, or, use the length of the segment $[a..b]$.

Figure 1 shows the archetype solutions for the *Seek* subclasses.

Seek & Memo submissions often contain replicate code because the operation of finding the last element of the initial segment and dropping the elements that belong to the initial segment is very similar. In the corresponding archetype solution, this is highlighted by using exactly the same pattern matches. *Seek & Memo* turns out to be the hardest to get right: except for one all submissions have run-time errors. The archetypes of the other subclasses all have the same structure, and differ only in the way the remainder of the input list of numbers is computed. This is also reflected in their names. The archetype solutions show how the standard higher-order library functions can be used for these. This is helpful for students, in particular when they have written custom functions for these computations.

```
SeekAndMemo    []        = []
SeekAndMemo    [a]       = [(a,a)]
SeekAndMemo    [a,b:cs]
| a+1 == b               = SeekAndMemo` a [b:cs]
| otherwise              = [(a,a) : SeekAndMemo [b:cs]]

SeekAndMemo` m []        = [(m,m)]
SeekAndMemo` m [a]       = [(m,a)]
SeekAndMemo` m [a,b:cs]
| a+1 == b               = SeekAndMemo` m [b:cs]
| otherwise              = [(m,a) : SeekAndMemo [b:cs]]

Seek [a]                 = a
Seek [a,b:cs]            = if (a+1 == b) (Seek [b:cs]) a

SeekAndFilter    []      = []
SeekAndFilter    [a]     = [(a,a)]
SeekAndFilter    [a:as]  = [(a,b) : SeekAndFilter (filter ((<) b) [a:as])]
where            b       = Seek [a:as]

SeekAndDropWhile []      = []
SeekAndDropWhile [a]     = [(a,a)]
SeekAndDropWhile [a:as]  = [(a,b) : SeekAndDropWhile
                                       (tl (dropWhile ((<>) b) [a:as]))]
where            b       = Seek [a:as]

SeekAndDrop      []      = []
SeekAndDrop      [a]     = [(a,a)]
SeekAndDrop      [a:as]  = [(a,b) : SeekAndDrop (drop (b-a+1) [a:as])]
where            b       = Seek [a:as]
```

Fig. 1. Archetype Seek solutions.

4.1.2 Bounds

In the *Bounds* solutions, the key design decision is to keep track of the lower and upper bound of the initial segment. The two main ways of doing this are

- *Bounds Values*: the recursive function keeps track of the current lower and upper bound value (initially, the head of the input list of numbers) and replaces the upper bound while successive numbers are encountered.
- *Bounds Indices*: the recursive function keeps track of the *index position* of the current lower and upper bound value (initially both are zero) and replaces the upper bound index while successive numbers are encountered.

Figure 2 shows the archetype solutions for the *Bounds* subclasses (the operation ($xs!!i$) selects the element at index i from list xs).

Bounds Values is the most frequently proposed solution to solve the *Segments* problem, and follows the programming pattern on lists that has been taught in the lecture. In a way, the *Bounds Values* solution is similar to the *Seek & Memo* solution, but in general results

```
BoundsValues []           = []
BoundsValues [a:as]       = BoundsValues‘ a a as

BoundsValues‘ a b []      = [(a,b)]
BoundsValues‘ a b [c:cs]
| b+1 == c                = BoundsValues‘ a c cs
| otherwise               = [(a,b) : BoundsValues‘ c c cs]

BoundsIndices []          = []
BoundsIndices as          = BoundsIndices‘ 0 0 as

BoundsIndices‘ a b xs
| b >= length xs-1        = [(xs!!a, xs!!(length xs-1))]
| 1+xs!!b == xs!!(b+1)    = BoundsIndices‘ a (b+1) xs
| otherwise               = [(xs!!a, xs!!b) : BoundsIndices‘ (b+1) (b+1) xs]
```

Fig. 2. Archetype Bounds solutions.

in a more elegant result because it has less code replication. The instructor can use the two archetype solutions to discuss the pros and cons of both approaches.

In the archetype solution of *Bounds Indices*, we have taken care that the guarded cases coincide with patterns and guards of the *Bounds Values* solution. Students who choose to use a *Bounds Indices* style of solution do not seem to feel comfortable with manipulating lists via structural recursion and pattern matching, and instead regard them more like arrays, which is a data structure they are very familiar with. We strongly discourage this programming style because it is much more prone to errors and steers the students away from learning to think about and program functions on recursive data structures in a correct functional way. All of the *Bounds Indices* submissions have a run-time error and 80% of the solutions print the output in the wrong format. In contrast, 68% of the *Bounds Values* submissions have a run-time error and 58% of the solutions print the output in the wrong format. The instructor can show and explain the *Bounds Values* archetype to explain how such a function should be structured.

Finally, we observe that list comprehensions and higher-order functions are hardly used in *Bounds*. No *Bounds Values* submission uses a list comprehension and only 16% use higher-order functions and 20% of the *Bounds Indices* use a list comprehension and no higher-order functions.

4.1.3 Fold

In the *Fold* solutions, the key design decision is that students notice that collecting segments has a *fold* structure. There are two subclasses in this high-level class structure:

- *Fold*: regardless of using a left-fold (*foldl*) or a right-fold (*foldr*), a glueing function is introduced that adds one number from the input list of numbers to the current partial solution. This number either extends the segment under construction or creates a new one.
- *Fold, Actually*: these are solutions that actually implement the fold program structure, but seemingly without realizing that this is the case.

```
Foldr  []      = []
Foldr  xs      = foldr Prepend [(last xs,last xs)] (init xs)

Prepend x [(a,b):segs]
| x+1 == a     = [(x,b):segs]
| otherwise    = [(x,x),(a,b):segs]

Foldl []       = []
Foldl [x:xs]   = foldl Append [(x,x)] xs

Append segs c
| b+1 == c     = init segs ++ [(a,c)]
| otherwise    = segs ++ [(c,c)]
where (a,b)    = last segs
```

Fig. 3. Archetype Fold solutions.

We only show the two archetype solutions of the *Fold* class in Figure 3 ($as ++ bs$ concatenates the lists *as* and *bs*; `last` ($xs ++ [x]$) returns x; `init` ($xs ++ [x]$) returns xs).

Surprisingly, half of the submissions of the *Fold, Actually* subclass do not use higher-order functions: the code is structured as a *fold* function, but instead of a higher-order function parameter an auxiliary function is used. Students who use the *Fold, Actually* are just one step away from adopting *Fold*, and, using their submission, the instructor can show them how to get there.

4.1.4 Count

In the *Count* solutions, the key design decision is to introduce a function that counts the number of elements n of the initial segment of an input list of numbers that has head element a. The following subclasses have been proposed:

- *Count & Drop*: the initial segment is $(a, a + n - 1)$, and the other segments are found recursively after dropping n elements from the input list of numbers.
- *Count & Continue*: the counting function memorizes a, so it can proceed recursively with the remainder of the list after determining n.
- *Zip Count & Drop*: the counting function exploits the property that a segment consists of successive elements, and the other segments are found recursively after dropping n elements from the input list of numbers.

Figure 4 shows the archetypes of the *Count* solutions.

Almost all *Count* submissions generate segments-as-strings on-the-fly (only two *Count & Drop* submissions do not). *Count & Drop* is similar to *Seek & Drop*. In the archetype solution, this is emphasized by giving the counting function the same structure as the seeking function. *Count & Continue* is similar to *Seek & Memo* in the decision to integrate counting the length of the initial segment with finding the remainder of the input list of numbers to continue with. The structure of *Count & Continue* is better because it has no code duplication. Finally, *Zip Count & Drop* is an interesting solution because it explicitly exploits the structure of a segment, viz. a sequence of successive numbers, using a list

```
CountAndDrop []          = []
CountAndDrop [a:as]      = [(a,a+n-1) : CountAndDrop (drop n [a:as])]
where n                  = Count [a:as]

Count [a]                = 1
Count [a,b:cs]           = if (a+1 == b) (1+Count [b:cs]) 1

CountAndContinue []      = []
CountAndContinue [a]     = [(a,a)]
CountAndContinue [a:as]  = CountAndContinue‘ a 1 as

CountAndContinue‘ a n [] = [(a,a+n-1)]
CountAndContinue‘ a n [b:bs]
| a+n == b               = CountAndContinue‘ a (n+1) bs
| otherwise              = [(a,a+n-1) : CountAndContinue [b:bs]]

ZipCountAndDrop []       = []
ZipCountAndDrop [a:as]   = [(a,last p):ZipCountAndDrop (drop (length p) [a:as])]
where p                  = [x \\ x <- [a:as] & i <- [0..] | x == a+i]
```

Fig. 4. Archetype Count solutions.

comprehension to elegantly denote this ([*f x y* \\ *x* <- *xs* & *y* <- *ys* | *p x y*] pairwise extracts elements *x* and *y* from lists *xs* and *ys*, accepting only those that satisfy predicate (*p x y*), and collecting them in the new list as values (*f x y*)).

4.1.5 Accumulate

In the *Accumulate* solutions, the key design decision is to use the accumulator pattern to keep track of all segments-under-construction. In this class, we also place hybrid solutions that accumulate only the initial segment (*AccumulateSegment*) or the initial segment and all segments-under-construction (*AccumulateSegments*). These three archetype solutions (Figure 5) represent the wide variety in solutions.

Accumulate is the purest application of the accumulator pattern. To emphasize the similarity with the *Foldl* solution, we use the same function *Append* (Figure 3) that either adds an element to the "current" segment or creates a new one. For students, it is instructive to see the relation between the accumulator pattern and the left-fold function (Section 4.1.3). *AccumulateSegment* is a hybrid of the accumulator pattern and the *BoundsValues* archetype solution (Section 4.1.2). *AccumulateSegments* separately accumulates the initial segment and all segments.

4.1.6 Segments

The *Segments* high-level class of solutions is interesting because the key design decisions show that students have focused on the key characteristic of a segment: they are a sequence of successive elements of which the first element has no predecessor and the last element has no successor in the input list of numbers. The subclasses are

- *Deflate Segments*: use a marker value to identify 'interior' elements of the segments, and use this to remove them from the input list of numbers.

```
Accumulate []             = []
Accumulate [x:xs]         = Accumulate‘ [(x,x)] xs

Accumulate‘ acc []        = acc
Accumulate‘ acc [x:xs]    = Accumulate‘ (Append acc x) xs

AccumulateSegment []      = []
AccumulateSegment [x:xs]  = AccumulateSegment‘ (x,x) xs

AccumulateSegment‘ s []   = [s]
AccumulateSegment‘ (a,b) [x:xs]
| b+1 == x                = AccumulateSegment‘ (a,x) xs
| otherwise               = [(a,b) : AccumulateSegment‘ (x,x) xs]

AccumulateSegments []     = []
AccumulateSegments [x:xs] = AccumulateSegments‘ [] (x,x) xs

AccumulateSegments‘ acc s [] = acc ++ [s]
AccumulateSegments‘ acc (a,b) [x:xs]
| b+1 == x                = AccumulateSegments‘ acc (a,x) xs
| otherwise               = AccumulateSegments‘ (acc ++ [(a,b)]) (x,x) xs
```

Fig. 5. Archetype Accumulate solutions.

- *Inflate Segments*: first turn every number in the input list of numbers into a singleton list and then concatenate all neighbors of which the last element of the first is the predecessor of the head element of the second segment.
- *Zip Predless Success*: pairwise zip the elements of the list of numbers from the input list that have no predecessors and the list of numbers from the input list that have no successors.

The archetype solutions are shown in Figure 6.

In *Deflate Segments* submissions, students stick to using a list structure of basic values and hence face the challenge to find a way to encode the segments structure in this list. A complicating factor is that list elements must have the same type, so all submissions resort to encoding the numbers as strings and using the marker symbol "-" to identify "interior" segment elements. Note that in the context of the assignment, this is a sensible choice because in the end segments need to be produced as a list of strings. However, the code quality suffers from lack of separation of concerns. The archetype solution uses marker value –1, but it could have been another value.

Exactly the opposite choice has been made in *Inflate Segments* submissions where students decide to immediately reshape the structure of the input list of numbers to a list of singleton lists, allowing them to elegantly express when two of these partial segments need to be merged into one.

In contrast with the previous two strategies, *ZipPredlessSuccess* submissions do not concentrate on "interior" elements at all but instead exploit the fact that segments are delimited by an input list element without an immediate predecessor (predless) and an input list element without an immediate successor (success). This results in an elegant solution of the *Segments* problem, by pairwise zipping the two lists.

```
DeflateSegments xs            = RemoveMarkers (InsertMarkers xs)

InsertMarkers []              = []
InsertMarkers [x]             = [x]
InsertMarkers [x,y:zs]
| x+1 == y                    = [x,-1 : InsertMarkers [y:zs]]
| otherwise                   = [x    : InsertMarkers [y:zs]]

RemoveMarkers [a,-1,b,-1,c:xs] = RemoveMarkers [a,-1,c:xs]
RemoveMarkers [a,-1,b:xs]     = [(a,b) : RemoveMarkers xs]
RemoveMarkers [a:xs]          = [(a,a) : RemoveMarkers xs]
RemoveMarkers []              = []

InflateSegments xs            = InflateSegments‘ (map (\x = [x]) xs)

InflateSegments‘ [s1,s2:segs]
| last s1+1 == hd s2          = InflateSegments‘ [s1 ++ s2 : segs]
| otherwise                   = [Extract s1 : InflateSegments‘ [s2 : segs]]
InflateSegments‘ [s]          = [Extract s]
InflateSegments‘ []           = []

Extract segment               = (hd segment, last segment)

ZipPredlessSuccess []         = []
ZipPredlessSuccess xs         = zip (predless, success)
where predless = [hd xs] ++ [a \\ pred <- xs & a <- tl xs | pred+1 <> a]
      success = [b \\ b <- xs & succ <- tl xs | b+1 <> succ] ++ [last xs]
```

Fig. 6. Archetype Segments solutions.

It is no coincidence that these subclasses differ greatly in terms of encountered issues. 80% of the *Deflate Segments* submissions have a run-time error and generate segments-as-string on-the-fly, but this is only 20% for the *Inflate Segments* submissions. Twenty-five percent of the *Zip Predless Success* submissions have a run-time error but have a better separation of concerns because no submission generates segments-as-string on-the-fly.

4.1.7 Exotic

The *Exotic* class is a somewhat loose collection of remaining solutions.

- *Les Uns Et Les Autres*: the input list of numbers is split into two sublists: one containing only the numbers that belong to singleton segments and one containing all other numbers.
- *Zip With Diff*: compute the list of differences between subsequent values in the input list of numbers and use these differences to determine whether an input number should be added to a current segment or constitutes a new segment.

In Figure 7, no archetype is shown for *Les Uns Et Les Autres* because of two reasons. The first reason is that a consequence of the design choice to try to simplify the problem fails to do this because the found singleton segments and non-singleton segments need to be merged correctly afterward (and indeed, all submissions forgot to do this). The second

```
ZipWithDiff []          = []
ZipWithDiff xs          = ZipWithDiff` [b-a \\ a <- xs & b <- tl xs] xs

ZipWithDiff` ds [x]     = [(x,x)]
ZipWithDiff` [d:ds] [x:xs]
| d == 1                = [(x,b):segs]
| otherwise             = [(x,x):(a,b):segs]
where
    [(a,b):segs]        = ZipWithDiff` ds xs
```

Fig. 7. Archetype Exotic solutions.

reason is that there is no advantage in handling non-singleton segments, and all of the proposed solutions can be identified by one of the earlier described archetype solutions. The *Zip With Diff* archetype solution shows that the choice of computing the differences between subsequent values helps to simplify the recursive code. The instructor can use this to compare this solution with the *Foldr* version (Figure 3) and let students try to find a version that corresponds with the *Foldl* version.

4.2 Code structure and quality

We make the following observations on code structure and code quality:

1. Except for two submissions that only sort the input list of numbers, all other solutions start with sorting and removing duplicates. These are functions that are readily available in the *Clean* prelude. Some students note that removing duplicates can be done efficiently after sorting and define their custom function to do this.
2. Ninety-five percent of the submissions compile successfully, the remaining 5% have errors that are trivial to solve. The six submissions with compiler errors are spread evenly over five high-level solution classes.
3. Sixty-four percent of the submissions show the segments correctly, the most frequent mistake is to emit singleton segments x as $x - x$.
4. Fifty-six percent of the submissions generate the segments-as-string on-the-fly instead of via a final pass on the generated segments. This occurs most frequently with submissions in the high-level solution classes *Count* (89%), *Seek* (77%), and *Accumulate* (73%).
5. Forty-eight percent of the submissions have run-time errors, of which the most frequent ones are not being able to handle an empty list of numbers as input, termination issues with the list of numbers, or not being able to handle the final segment being a singleton segment. These submissions were likely only tested on the example data of the assignment (a large CD collection). Often, these errors take little effort to fix because they have a single missing pattern match in the function that traverses the list structure.
6. Thirty-two percent of the submissions use higher-order functions. Of course, all *Fold* solutions use higher-order functions. The only two other subclasses that attract a high percentage of higher-order functions are *Zip With Diff* (one solution) and *Inflate Segments* (four out of five).

7. One design decision that surprised the author is that in 7% of the submissions, the list of numbers is converted to a list of string representations of these numbers, typically in the initial cleaning phase of the program (sorting and removing duplicates). This complicates the remainder of the program for several reasons. The first reason is that strings need to be converted to and from integers when finding the segments, which clutters the code and is not efficient. The second reason is that sorting string representations of numbers yields an ordering different from sorting the respective numbers. Eleven percent of all submissions did not return the list of segments in the correct order, and these contained all submissions that worked with string representations of numbers immediately.

   The instructor can use this observation to discuss this design decision with the students and guide them to improve their code structure, and, in this case, obtain correct results.

It should be noted that the above discussion is not an attempt to reflect the actual feedback given by the teaching assistants. Despite the above observations concerning efficiency of code, in the 2016–2019 editions of the course, the focus is on correctness and program structure rather than efficiency. We observe that failing to show the segments correctly (issue 3) or generating segments-as-string on-the-fly (issue 4) both have a strong correlation with the odds of having a run-time error (0.89 and 0.82, respectively). The instructor can use this to demonstrate that choosing a good program structure pays off. In case of the *Segments* problem, a good structure is to split the solution into three subsequent steps: first, the cleaning step removes duplicates and sorts the input list of numbers (as noted above, virtually all students have used this approach), second, the segments are identified, either via boundary values or lists of values, and third, turn the found segments into string representations.

5 Discussion

It is interesting to see how the results of this study carry over to other functional programming courses. Uncovering high-level solution classes and its subclasses depends on the programming language concepts and techniques, the set of student submissions, and their interpretation by the instructor. By concentrating on finding the dominant idea of submissions, we expect that similar high-level solution classes will be unveiled.

The *Segments* problem is a list processing problem, but because it has two reshaping components it is less suitable as an exercise immediately when lists and list comprehensions have been instructed (Table 1). This is the main reason to put it forward when higher-order functions have been instructed. Students pick up the opportunities offered by higher-order functions, because 32% of the submissions incorporate higher-order functions.

The archetypes highlight the essence of a solution. The instructor can use them to discuss the design space of the *Segments* problem with students, how solutions relate to each other, and if one solution can be considered to be better than another. The diagram in Figure 8 illustrates this. Archetype solutions that belong to the same high-level solution strategy have the same color. An edge between archetype solutions means that they are related, as presented in the previous section. The instructor can use these to show students how to apply a particular technique and compare it to its alternative, without stating that one way is better than another. However, in cases where we argue that one

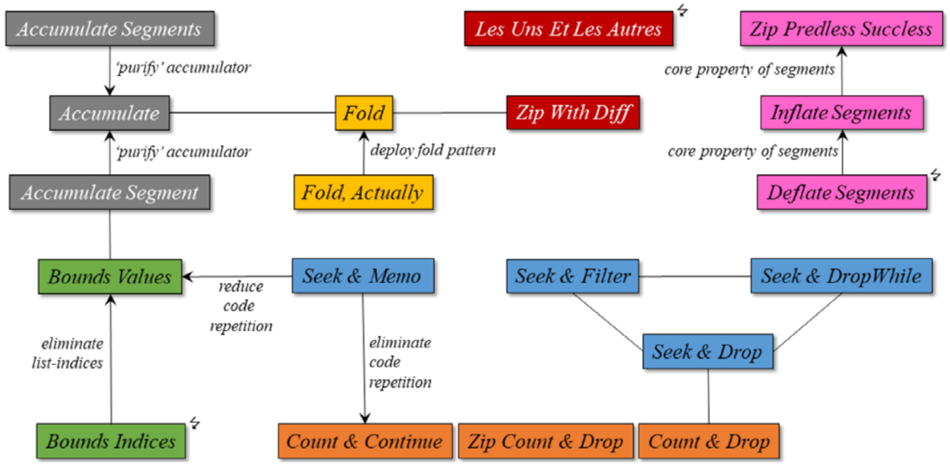

Fig. 8. Relating the archetype solutions.

archetype solution is better than another, a directed edge toward the better solution is used, annotated with a brief characterization of the improvement. Solutions marked with ϟ are solutions that we advise students to avoid.

Of the *Seek* solutions, *Seek & Memo* is the hardest to get right, because in contrast with the other *Seek* solutions, the student writes a seek function that is not committed to the single task of finding the last element of the initial segment. Using these other solutions, and *Bounds Values* and *Count & Continue*, the student can see how to improve their solution by improving on the separation of concerns.

Of the *Fold* solutions, half of the solutions do not use one of the standard *fold* functions. This provides an excellent opportunity for the instructor to stay close to their submission and refactor it into a *fold*-based solution. This is also the case for the *Accumulate* submissions.

Almost all *Count* submissions generate segments-as-strings on-the-fly. We do not know why this is so strongly the case for this class of solutions. Submissions that generate segments-as-strings on-the-fly should be avoided because it is indicative of lack of separation of concerns and there is a strong correlation with the odds of having a run-time error.

The “to be avoided” solutions provide opportunities to the instructor to help students forward. In case of *Bounds Indices* submissions, students manipulate lists as if they are arrays, so they can be helped with practice material on lists. In case of *Les Uns Et Les Autres* submissions, the instructor can enter a constructive dialog with the student to determine the used problem analysis and guide them toward an alternative solution. Finally, *Deflate Segments* submissions attempt to avoid reshaping the list, so they can benefit from more exercises with one or more reshaping components.

6 Conclusions

We put forward the *Segments* problem as a tool to discover which solution strategies students in functional programming adopt. The instructor can use *archetype* solutions to identify and discuss the key design decisions that have been made by the students. These traits make the *Segments* problem a welcome addition to the tool kit of instructors to assess the analytic skills of their students in functional programming. The analysis of the *Deflate Segments* archetype solution suggests to rephrase the *Segments* problem in such a way that the input list of numbers consists of *positive* numbers only. This will hopefully prevent students from choosing to generate segments-as-strings on-the-fly.

Acknowledgments

Johan Jeuring brought the *Functional Rainfall* problem to the author's attention at the 4th International Workshop on Trends in Functional Programming in Education, TFPIE 2015. The author thanks the participants of TFPIE 2020 for their constructive feedback on the initial presentation of this investigation, and in particular Jurriaan Hage for suggesting to eliminate "identical" solutions to obtain a clear view about the relative contributions of the high-level solution strategies. Finally, the author thanks the reviewers for their constructive and valuable feedback, which has greatly improved the text.

Conflicts of Interest

None.

References

Achten, P. (2011). The soccer-fun project. *JFP* **21**(1), 1–19, doi: 10.1017/S0956796810000055.

Fisler, K. (2014). The recurring rainfall problem. In *ICER 2014*, August 11–14 2014, Glasgow, Scotland, UK, ACM. 978-1-4503-2755-8/14/08, http://dx.doi.org/10.1145/2632320.2632346.

Fisler, K., Krishnamurthi, S. & Siegmund, J. (2016). *Modernizing Plan-Composition Studies, CIGCSE'16*, March 2–5 2016, Memphis, TN, USA, pp. 211–216, https://doi.org/10.1145/2839509.2844556.

Hutton, G. (2002). The countdown problem. *JFP* **12** (6), 609–616, DOI: 10.1017/S095679680100430.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, **10**(8), 707–710.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, **29**(9), 850–858.