Segments: a better Rainfall problem for FP

position presentation TFPIE'20 (12-2-2020)

Peter Achten

Radboud University, Nijmegen, Netherlands

P.Achten@cs.ru.nl

---

note: this is the same version as shown at TFPIE'20 except that all concrete code fragments have been removed

What this talk is about

- Brought to my attention by Johan Jeuring (TFPIE'15)
  - https://wiki.tfpie.science.ru.nl/TFPIE2015
- Elliot Soloway's Rainfall problem (1986)
  - assessing student progress in learning to construct programs
- Kathi Fisler (2014)
  - how about students in functional programming?
- My position:
  - Segments is more suited for functional programming than Rainfall

The Rainfall problem

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

[Soloway, 1986]

- experiments mostly done for imperative programming languages
- I/O early
- limited exposure to data structure, typically only arrays

"what do students from functional-first CS1 courses do with Rainfall?"

[Fisler, 2014]

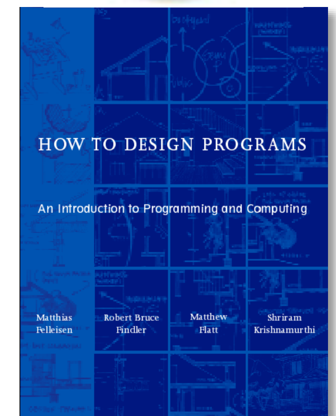
The Rainfall problem

"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."

[Fisler, 2014]

- no I/O
- input is list of numbers
- no suggestions of functional programming patterns

Context



- Programming language Racket
- How to Design Programs (HTDP)
  - structured design methods
  - bottom-up first, top-down second
- Language features:
  - atomic values, (nested) structures, lists, trees
  - functions, conditionals, user-defined structures
  - naming intermediate computations
  - higher-order functions: filter, map
- Experiments:
  - exam question, supervised lab assignment, out-of-class time
  - 207 solutions analyzed

High-level program structures

```
while not sentinel
   if non-negative number
   then incr count,
           add to sum
if count is positive
then report average
else report 'no data'
```

Single Loop (19%)

```
while not sentinel
   if non-negative number
   then add to new list
if length is positive
then count,
       sum,
       report average
else report 'no data'
```

Clean First (42%)

```
while not sentinel
   if non-negative number
   then incr count
while not sentinel
   if non-negative number
   then add to sum
if count is positive
then report average
else report 'no data'
```

Clean Multiple (21%)

… + attempt to adjust the results of sum and count to handle negative and sentinel

Clean After (1%)

… + neglect non-negative numbers and sentinel, but count, sum, report average

No Cleaning (3%)

… + mangled or unstructured solution

Unclear (14%)

High-level program structures

```
take elements while not sentinel
filter non-negative elements
if empty
then report ‘no data’
else count,
       sum,
       report average
```

this solution seems to be missing, why?

Low-level errors

- **Single Loop** (19%) has least amount of errors:
  - 16% of solutions fail to ignore non-negative values
  - 16% of solutions encounter division-by-zero error
  - 74% zero errors, 21% one error, 5% two errors
- **Clean First** (42%) comes second:
  - 18% of solutions fail to ignore non-negative values
  - 36% of solutions encounter division-by-zero error
  - 66% zero errors, 22% one error, 7% two errors, 6% three-six errors
- **Clean Multiple** (21%) most suspect to errors:
  - 58% of solutions fail to ignore non-negative values
  - 52% of solutions encounter division-by-zero error
  - 46% zero errors, 23% one error, 13% two errors, 15% three-six errors

Observations

- functional rainfall problem gives rise to 3 archetype solutions
- most students get the high-level program structure right (82%)
- functional rainfall solutions have a very small low-level error rate (at most 2 errors in 100%, 95%, and 82% of archetype solutions)

Position statement

- functional rainfall is neither sufficiently challenging nor differential

The Segments problem

"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 showing all individual elements they are shown 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 are shown as strings."

Key differences with Rainfall:

- requires more than straightforward list traversal functions
- allows a greater variety of high-level program structures of choice

Context

- Functional Programming for AI course (3 ects)
- Clean programming language
- 2nd year students, mandatory course
  - 1st year Java programming (2×6 ects)
- 7 weeks:
  1. introduction (basic types, tuples, reduction, functions, patterns, guards)
  2. types (inference, polymorphism, synonyms, records, algebraic types)
  3. overloading (modules, type classes)
  4. lists (recursion, reduction, overloading, list comprehensions)
  5. higher-order functions (currying, λ-abstraction, composition, folds)
  6. reasoning (reduction strategies, referential transparency, equational reasoning)
  7. case study (Countdown – Graham Hutton)

Context

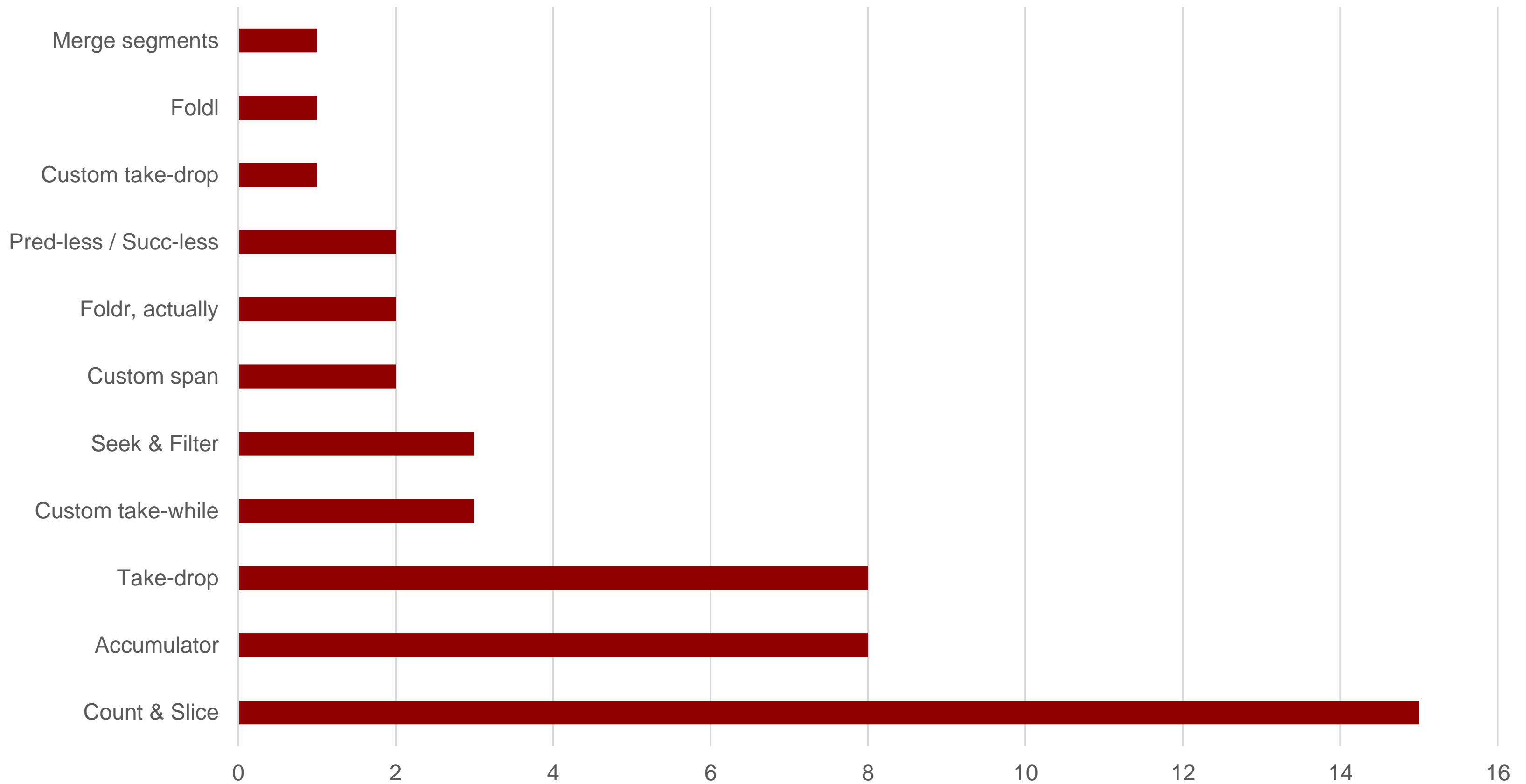
- Functional Programming for AI course (3 ects)
- Clean programming language
- 2nd year students, mandatory course
  - 1st year Java programming (2×6 ects)
- 7 weeks:
  1. introduction (basic types, tuples, reduction, functions, patterns, guards)
  2. types (inference, polymorphism, synonyms, records, algebraic types)
  3. overloading (modules, type classes)
  4. lists (recursion, reduction, overloading, list comprehensions)
  5. higher-order functions (currying, λ-abstraction, composition, folds)
  6. reasoning (reduction strategies, referential transparency, equational reasoning)
  7. case study (Countdown – Graham Hutton)

Context

- lab assignments are formative, duo programming (some solo)
- segments problem is last part of the assignment, formulated in the context of that assignment (querying a large music data base)
- 62 solutions of 2018 (still to analyze: 67 solutions of 2019)

- 16 out of 62 submissions invalid solutions:
  - 3 simply don’t do the segment problem
  - 6 only clean the data (remove duplicates, sorting)
  - 2 can’t handle singleton segments
  - 5 attempts are unfeasible
- sample solutions have been edited for readability

High-level program structures



High-level program structures

1. **Count & Slice**(33%)
   - use a counter to identify segment, slice and collect as string-segment
2. **Accumulator**(17%)
   - accumulating segments, only current segment, current segment & element, current segment and segments, current segment and its start-end
3. **Take-drop**(17%)
   - use head element to take segment and generate string-segment, drop to recurse
4. **Custom take-while**(6.5%)
   - use custom take-while to obtain segment, eliminate elements to recurse
   - map to create the string-segments
5. **Seek & Filter**(6.5%)
   - search end of segment of head value, turn into string-segment
   - recurse with list of numbers filtered by values larger than end of segment
6. **Custom span**(4%)
   - search last element of current segment and return remainder of number list
   - accumulate string-segments

High-level program structures

7. **Foldr, actually**(4%)
    - explicit recursive function that captures foldr
    - map to create string-elements
8. **Pred-less / Succ-less**(4%)
    - zip list of predecessor-less elements with successor-less elements
    - map to get string-segment
9. **Custom take-drop**(2%)
    - use zip list-comprehension to determine segment elements, generate string-segment, drop to recurse
10. **Foldl**(2%)
    - application of foldl to construct segments
    - map to create string-segments
11. **Merge segments**(2%)
    - turn every number into singleton list, merge subsequent lists
    - map to create string-segments

Observations

- Segments gives rise to more archetype solutions than Rainfall
- They are also quite different from each other

Count & Slice (33%)

- use a counter to find the number of elements of initial slice
- once found, collect segment by taking slice
- continue with dropping segment

- code is driven by the guards
- it has an ‘operational’ feel

Accumulator(17%)

- use an accumulator of segments
- per list element, decide where to add the element
- when all numbers are exhausted, return the accumulator

- in this case the segment accumulator is relatively easy to follow, but much more complicated versions have been proposed

Take-drop[(17%)]

- one function that keeps taking elements that belong to initial segment
- one function that keeps dropping elements that belong to initial segment
- combine to collect all segments

- despite of the verbosity, each function has a clear task
- naming can be improved by student

Foldr, actually (4%)

- the structure of the function is identical to foldr
- completely dedicated to this particular problem

- this student actually implemented foldr
- can easily improve code to reflect this

Seek & Filter(6.5%)

- use an incrementing value to detect final value of initial segment
- use filter to keep all elements that do not belong to initial segment, and recurse

---

- each function has a clear task
- code concentrates on correctness, but is rather inefficient
- nice follow-up task for student is to improve performance

Pred-less / Succ-less$^{(4\%)}$

- strike out all elements that have no predecessor
- strike out all elements that have no successor
- zip to pair them and obtain all segments

- very neat idea!
- rather inefficient
- nice follow-up task for student is to improve performance

Merge segments(2%)

- turn all numbers into singleton lists
- concatenate if last element is predecessor of head of next element

---

- very neat idea!
- rather inefficient
- nice follow-up task for student is to improve performance

Observations

- All solutions first clean the data set (remove duplicates, sort)
  - standard library functions; a few solutions re-implement them
- Many solutions generate string-segments on-the-fly
  - separation of concerns can be improved in these cases
- Many solutions are inspired by standard list functions
  - drop, take, slice, filter, takeWhile, span
- Count & Slice (33%) solutions have an ‘operational feel’
- Accumulator (17%) solutions tend to become cluttered
- Take-drop (17%) solutions resemble list consumer approach

Observations

- Segments problem gives rise to at least 17 archetype solutions
- most students get the high-level program structure right (46 out of 62 ≅ 74%)
- most 'popular' archetype solutions are Count & Slice, Accumulator, Take-drop (67%)

Position statement

- the Segments problem is a solution-richer problem than Rainfall
- the chosen high-level program structure can be used to give the student guidance for further study, thus improving their understanding of FP