185.A05 Advanced Functional Programming SS 2020

Monday, 27 April 2020

Assignment 4

on Chapter1 thru Chapter 3, Chapter 8

Topics: Algorithm Patterns (Generate/Select, Greedy, Backtracking), Abstract Data Types (used by some algorithm patterns)

Submission deadline: Monday, 11 May 2020, 12am

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Important:

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 1. If you have any questions regarding these instructions, ask your questions in the TISS forum. Following these instructions is paramount to ensure a smooth processing of your submitted file with the test system.
2. Store all functions to be written for this assignment in a top-level file named

   `Assignment4.hs`

   of your group directory starting with the module declaration

   `module Assignment4 where`

   Comment your program meaningfully; use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 4.
3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment4.hs`. Import declarations for self-defined modules will fail: Only `Assignment4.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.
4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Programming tasks:

In ancient Egypt, fractional numbers $\frac{a}{b}$, $a < b$, were written as sums of pairwise disjoint unit fractions (Stammbrüche), i.e., as sums of pairwise disjoint fractional numbers where the numerators (Zähler) are 1 and the denominators (Nenner) are strictly positive integers, e.g.:

$$\frac{2}{3}=\frac{1}{2}+\frac{1}{6}, \qquad \frac{2}{5}=\frac{1}{3}+\frac{1}{15}, \qquad \frac{3}{7}=\frac{1}{3}+\frac{1}{11}+\frac{1}{231}$$

Usually, the representation of a fractional number as a sum of pairwise disjoint unit fractions is not unique, e.g.:

$$\begin{aligned}\frac{9}{20} &= \frac{1}{4}+\frac{1}{5}\\ &= \frac{1}{3}+\frac{1}{9}+\frac{1}{180}\end{aligned}$$

$$\begin{aligned}\frac{5}{31} &= \frac{1}{8}+\frac{1}{30}+\frac{1}{372}+\frac{1}{3720}\\ &= \frac{1}{7}+\frac{1}{55}+\frac{1}{3979}+\frac{1}{23744683}+\frac{1}{1127619917796295}\\ &= \frac{1}{7}+\frac{1}{56}+\frac{1}{1736}\end{aligned}$$

The two examples show that different representations of a fractional number as sums of pairwise disjoint unit fractions can be differently appealing.

Intuitively, the smaller the number of summands and the smaller the denominators of the unit fractions are, the more appealing a representation is but the longer it might take to find some.

The examples show that these goals are conflicting and can usually not met at the same time. Therefore, we want to implement and experiment with different strategies for computing representations of a given fractional number as sums of pairwise disjoint unit fractions.

Note: Values of type `MaxDenominator` are used to constrain the search space to ensure termination of the strategies, where necessary. `MaxDiff` and `MaxSummands` values will be used to define some of the selection criteria for representations. Representations of fractional numbers, finally, i.e., sums of unit fractions are represented by ascending lists of natural numbers being the denominators of the unit fractions of the representation.

```
type Nat1        = Integer  -- Natural numbers starting with 1: [1..]
type Numerator   = Nat1
type Denominator = Nat1
type FracNum = (Numerator,Denominator) -- Only pairs with numerator < denominator
type MaxDenominator = Nat1
type MaxDiff        = Nat1
type MaxSummands    = Nat1
type Representation = [Denominator]
```

1. *Greedy search.* Greedy search picks in each step the largest unit fraction smaller than or equal to the fractional number it is applied to in the current step. Use the algorithm pattern for *greedy search* of Chapter 3 to implement `greedy`:

```
greedy :: FracNum -> Representation
greedy ...

greedy (5,31) ->> [7,55,3979,23744683,1127619917796295]
```

(Note: The greedy pattern of Chapter 3 yields a list of results. Here, however, this list will always be singleton, i.e., you will have to unpack the one and only entry from this list to get the result of `greedy`.)

2. *Generate/select search.* `gs1` and `gs2` shall first generate all candidate representations (using a generator), picking then the one (or: the ones) matching the selection criterion under consideration (using a selector). In detail:

   2.2 The generator `gen` generates all representations of a given fractional number as sums of unit fractions, where the size of denominators occurring is limited (i.e., smaller or equal to) by the value of the `MaxDenominator` argument. These are called the *candidate representations* or just *candidates.* Note: Due to constraining the search space, there might be no valid representation at all. In this case, the result list of `gen` shall be the empty list. If there is more than one representation matching the selection criterion, no particular order of the entries of the result list is required.

   ```
   gen :: FracNum -> MaxDenominator -> [Representation]
   ```

   2.2 `gs1` picks among all candidates (cf. 2.1) those with the smallest number of summands. If there is more than one candidate matching this criterion, no particular order of the entries of the result list is required. Note: Due to constraining the search space, there might be no solution at all. In this case, the result shall be the empty list:

   ```
   gs1 :: FracNum -> MaxDenominator -> [Representation]
   ```

   2.3 `gs2` picks among all candidates (cf. 2.1) the one (and only one) with the largest denominator occuring being the smallest one of the largest ones of all candidates. Note: Due to constraining the search space, there might be no solution at all. In this case, the result shall be the value `Nothing`:

   ```
   gs2 :: Frac -> MaxDenominator -> Maybe Representation
   gs2 ...

   gs2 (5,31) 4200 ->> Just [7,56,1736]
   gs2 (5,31) 42   ->> Nothing
   ```

3. *Backtracking search.* Backtracking search systematically explores the search space until all solutions are found. Use the algorithm pattern for *backtracking search* of Chapter 3 to implement `bt1` and `bt2`. In detail:

   3.1 `bt1` picks among all representations of the search space those where the largest denominator occuring in the representation is smaller or equal to the `MaxDenominator` value and the difference of the largest and the smallest denominator occuring in the representation is smaller or equal to the

MaxDiff value. If there is no such representation, the result list shall be the empty list.

```
bt1 :: FracNum -> MaxDenominator -> MaxDiff -> [Representation]
```

3.2 `bt2` picks among all representations of the search space those where the largest denominator is smaller or equal to the `MaxDenominator` value and the number of summands is smaller or equal to the `MaxSummands` value. If there is no such representation, the result list shall be the empty list.

```
bt2 :: FracNum -> MaxDenominator -> MaxSummands ->[Representation]
```

4. **Without submission:**

4.1 Test all implementations with values of your own choice.

4.2 Compare the performance and scalability of all implementations.

4.3 (Re-) implement `bt1` and `bt2` using generate/select search.

4.4 Is backtracking an appropriate algorithm pattern to (re-) implement `gs1` and `gs2`? Why? Or, why not?

4.5 From a cognitive point of view, do you have a preference for the generate/select or the backtracking implementations regarding ease of implementation, comprehensibility, etc.? Why? (Note, there is no 'right' or 'wrong' to this question in an absolute sense).

4.6 Does one of the strategies yield 'on average' the most appealing representation(s)?

4.7 How is the 'technique' called used for defining the range type of `gs1`, `bt1` and `bt2`? (Cf. Chapter 15 and 16).

4.8 Are their fractional numbers allowing `gs1` to yield nontrivial result lists with two or more entries? Try to find such fractional numbers, or to prove that there are none.

4.9 What could be other reasonable and worthwhile search strategies to find appealing representations or selection criteria to pick particular representations as solution?

4.10 Implement (some of) these strategies and selection criteria you consider particularly promising. Do they meet your expectations when experimenting with their implementations? If not, can you explain why they possibly fail to meet your expectations? Based on your analysis, can you improve the implementations to meet your expectations? If not, why?

4.11 Think about the computational complexity of all strategies both in theory (in terms of 'big-O' $\mathcal{O}(...)$) and practice (in terms of performance and scalability for given fractional numbers getting more and more complex to decompose; how can 'getting more and more complex' be formalized?).

4.12 *Generate/select search.* Consider implementing and experimenting with `gs3` and `gs4`. Do they work?

4.12.1 `gs3` picks among all candidates (cf. 2.1) the one greedy search would deliver. Note: Due to constraining the search space, there might be no solution at all. In this case, the result shall be the value `Nothing`:

```
gs3 :: FracNum -> MaxDenominator -> Maybe Representation
gs3 ...

gs3 (5,31) 127619917796295
                ->> [7,55,3979,23744683,1127619917796295]
gs3 (5,31) 42 ->> Nothing
```

4.12.2 `gs4`: Same as `gs4` but without constraining the search space; it is important that the generator used by `gs4` is *fair*, i.e., every possible representation candidate must be computed within a finite amount of time/steps.

```
gs4 :: FracNum -> Representation
gs4 ...

gs4 (5,31) ->> [7,55,3979,23744683,1127619917796295]
```

(Note: If working, the performances of `gs3` and `gs4` might be too bad to compute the result of `gs3 (5,31) 127619917796295` and `gs4 (5,31)`, respectively, in reasonable time.)

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller