CMPT 383: Vitamin #01

Anders Miltner
miltner@cs.sfu.ca

Introduction

This Vitamin is here to get you accustomed to programming in Haskell. In this Vitamin, you will practice basic coding Haskell control flow, practice list processing in Haskell, and practice simple recursion.

This submission will be autograded. There are some portions of the assignment that are ungraded, and some that will be graded. We provide a (partial) test suite for partial validation. You can run these tests by opening a terminal in the v1 directory, and running `stack test`.

In this, and all future assignments, carefully read the instructions. Follow all instructions provided carefully, not following the instructions may give you a zero on the assignment.

In this, and all future assignments, do *NOT* edit the imports. If the imports are any different from what is provided, you may get a zero on the assignment. The imports from the assignment have been selected to give you access to some funcions, but not access to some others.

However, unless it otherwise specified, you are allowed to use any function that does not require adding an import. Basically, as long as you do not touch imports, you will not have any points removed.

Do not use list comprehensions in this vitamin. Doing so will give you a zero on the vitamin.

Additionally, in this, and all future assignments, do *NOT* edit the types of the provided functions unless explicitly instructed to. Changing the types of the functions will almost certainly break all the tests, giving you a zero.

Finally, in this, and all future assignments, ensure that your code compiles. If your code does not compile on the test suite, you will get a zero on the assignment. Your code will compile if it compiles locally, and you didn't change any of the types unless explicitly instructed to.

1 List-based GCD Algorithm

In this section, you will write a list-based algorithm for finding the greatest common divisor of two numbers. This will provide a place to practice recursion, manipulating integral numbers, list processing, and fleshing out a test suite. In this problem, you will only be handed natural numbers. In other words, you don't need to worry about being provided inputs of 0 or negative numbers.

Open the file "Gcds.hs". In this file, you should see a number of stubbed-out functions – `isDivisor`, `allDivisors`, `listIntersection` and `listGcd`.

1.1 `isDivisor`

First you should implement `isDivisor`. Calling `isDivisor a b` should return true exactly when there exists some integer k such that $a * k = b$. A way to quickly check this is by determining whether $b\%a == 0$. In Haskell the mod function is just a function: `mod`. The code: `mod a b` returns the same as most languages `a % b`, or what is commonly known in English as "a mod b". Once you've implemented isDivisor, test it! We have provided a benchmark suite of a few tests to check your code with. However, this benchmark suite is non-exhaustive. Try adding some of your own!

1.2 `allDivisors`

Next, given a number, we want to get all divisors from that number. We want those numbers returned from largest to smallest. For example, `allDivisors 8 = [8,4,2,1]`.

1.3 `listIntersection`

Almost done! Now we would like to figure out the intersection of two lists. In `listIntersection`, you can assume that the incoming lists are ordered from largest to smallest. You should output a list also ordered from largest to smallest. For example, if you intersect the same list with itself, you should get that list back!

1.4 `listGcd`

Last bit! From here, you can find a list of all common divisors by intersecting the list of a's divisors with the list of b's divisors. This list should be ordered from largest to smallest, so you just need to extract the first from it!

2 Purely Functional Queue

A queue is a fundamental data structure, and is critical for infrastructure like packet processing, CPU scheduling, network routers, and more.

All of these applications require incredibly efficient processing. Having O(1) enqueing and dequeing is critical in such situations.

Typically, these queues are encoded with a doubly-linked list. When an element is enqueued, it is added to the front end of the list, and when an element is dequeued, it is removed from the back end of the list. This is possible in O(1) because references are maintained to the front and back ends of the list.

However, references and doubly-linked lists are both impossible in functional languages like Haskell. So we must use a different approach. In Haskell, rather than using a doubly-linked list, we will use two interacting lists. By doing this, we can achieve purely functional queues, while maintaining (amortized) O(1) enqueues and deques.

To achieve this, you will write three operations: `empty`, `enqueue`, and `dequeue`. Empty will create an empty queue. Enqueue will add an element to the queue. Dequeue will remove (and return) the next element to be dequeued.

Intuitively, the way this queue is implemented is that there is a "popping" list and a "pushing" list. When elements from the "popping" list are exhausted, the "pushing" list is reversed, and becomes the "popping" list, and the pushing list is set to empty.

Open the file "Queue.hs". In this file, you should see a number of stubbed-out functions – `empty`, `enqueue`, and `dequeue`.

2.1 `empty`

We've defined our "Queue" type to be a pair of two lists of integers. One of these lists is the "popping" list, and the other is the "pushing" list. For now we just want to create an empty queue.

To create a pair of lists, where the first element of the pair is `l1`, and the second is `l2`, one writes `(l1,l2)`.

To extract the first list in a queue `q`, one calls `fst q`. To extract the second, one calls `snd q`. So, `fst (l1,l2) = l1`, and `snd (l1,l2) = l2`.

You can also pattern match pairs! `f (l1,l2) = ...` would pattern match the first list of the pair into the variable `l1`, and pattern match the second list of the pair into `l2`.

2.2 `enqueue`

Now you must choose one of the lists to be the "pushing" list. When you enqueue an element, you simply prepend it to the front of that list!

2.3 `dequeue`

Dequeing is sometimes easy, and sometimes hard.

The easiest case is when the queue is empty. When the queue is empty, you should return the pair `(0,empty)`. In other words, popping empty returns the popped element 0, and another empty queue.

The next easiest case is when the "popping" list is nonempty. All you have to do is remove that element! So, You should return the head of the popping list, and create a new queue with the same pushing list, where the popping list has the head removed. You then want to return a pair consisting of the new queue, and the popped element. If `e` is the popped element, and `q` is the new queue, the pair that returns both is `(e,q)`.

The hard case is where the "popping" list is empty, but the "pushing" list is nonempty. At this phase, we take the following steps:

1. Convert the pushing list into the new popping list. However, remember that the head of the pushing list is the most recently added element. That means we want it to be the last element of the popping list (which is processed front-to-back). So, the new queue should have two lists: the pushing list – which should be empty, and the popping list, which should be the reverse of the old pushing list.

2. Call dequeue on this updated queue.