Computer Science 3MI3 – 2020 homework 4

Working with lazy (potentially infinite) lists in Scala

Mark Armstrong

October 7th, 2020

Contents

Introduction

Preamble

Throughout this course, we will investigate many evaluation/parameter passing methods. Specifically, we begin to investigate this topic during our discussion of reduction strategies in the untyped -calculus.

Two ways to classify evaluation strategies are as either

- *eager* (sometimes called strict),
  - performing evaluation as soon as possible, or
- *lazy* (sometimes called non-strict),
  - delaying evaluation until absolutely necessary (if it is needed at all).

Most programming languages employ eager evaluation in most circumstances. Notable exceptions include:

- "Short-circuit" operations, such as boolean and/or operations.
  - For instance, a short-circuit and operation evaluates one argument (usually the first), and if it is false, simply returns false, without checking the second argument.
- (Conditional) branching constructs such as `if-then-else`, `switch` statements and pattern matching.

  - The statements/expressions of the branches not taken are not evaluated.

- Function definitions.

  - Even if the argument type of the function is `Unit`, so there is only one possible argument, functions are never evaluated until called (invoked.)

This last we can make use of to create our own infinite datatypes in any language which supports higher-order functions (functions which can take other functions as arguments) and algebraic datatypes (which can take functions as parameters).

A type for this assignment

(In Friday's lecture, we stated that we would not construct our own "infinite" datatypes; this was revised during the tutorial, and in this homework, we work with the following new type.)

Consider the type

```
sealed trait Stream[+A]
case object SNil extends Stream[Nothing]
case class Cons[A](a: A, f: Unit => Stream[A]) extends
↪  Stream[A]
```

of *potentially* infinite lists. (If a `Stream` ends with a `SNil`, it is finite; but this is not always the case, as streams may be defined recursively so that `SNil` is not part of the stream.)

Refer to the "Infinite data in Scala" lecture notes, available as

- HTML,
- PDF, or
- plaintext Org source,

for more explanation of this type and some method definitions for it.

In particular, you will likely want to use the `take` method to be able to print portions of your streams.

Boilerplate

Submission procedures

Submission method

Homework should be submitted to your McMaster CAS Gitlab respository in the `cs3mi3-fall2020` project.

Ensure that you have **pushed** the commits to the remote repository in time for the deadline, and not just committed to your local copy.

Naming requirements

Place all files for the homework inside a folder titled `hn`, where `n` is the number of the homework. So, for homework 1, use the folder `h1`, for homework 2 the folder `h2`, etc. Ensure you do not capitalise the `h`.

Unless otherwise instructed in the homework questions, place all of your code for the homework in a single file in the `hn` folder named `hn.ext`, where `ext` is the appropriate extension for the language used according to this list:

- For Scala, `ext` is `sc`.
- For Prolog, `ext` is `pl`.
- For Ruby, `ext` is `rb`.
- For Clojure, `ext` is `clg`.

If multiple languages are used in the homework, submit a `hn.ext` file for each language.

If the language supports multiple different file extensions, you must still follow the extension conventions above.

Incorrect naming of files may result in up to a 10% deduction in your grade.

Do not submit testing or diagnostic code

Unless you are instructed to do so in the homework questions, **you should not submit testing code with your homework submission**.

This includes

- any `main` function,

- any `print` statements which output information **that is not directly requested as console output in the homework questions**.

If you do not wish to remove diagnostic print statements manually, you will have to find a way to ensure that they disabled in your final submission.

For instance, by using a wrapper on the print function or macros.

Due date and allowance for technical difficulties

Homework is due on the second Sunday following its release, by the end of the day (midnight). Submissions past 00:00 may not be considered.

If you experience technical difficulties leading up to the submission time, please contact Mark **ASAP** with the details of the problem and, if possible, attach the current state of your homework to the communication. This information will help ensure we are able to accept your submission once the technical difficulties are resolved.

Proper conduct for coursework

Individual work

Unless explicitely stated in the homework questions, all homework in this course is intended to be *individually completed*.

You are welcome to discuss the content of the homework in the public forum of the class Microsoft Teams team homework channel, though obviously solutions or partial solutions should not be posted or described.

Private discussions about the homework cannot reasonably be forbidden, but such discussions should follow the same guidelines as public discussions.

Inappopriate collaboration via private discussions which is later discovered by course staff may be considered academic dishonesty.

When in doubt, make the discussion private, or report its contents to the course staff by making a note of it in your homework.

To clarify what is considered appropriate discussions of homework content, here are some examples:

1. Discussing the language features introduced or needed for the homework.
    - Such as relevant builtin datatypes and datatype definition methods and their general use.

- Code snippets that are not partial solutions to the homework are welcome and encouraged.

2. Questions of the form "What is meant by `x`?", "Does `x` really mean `y`?" or "Is there a mistake with `x`?"

   - Of course, questions of those form which would be answered by partial solutions are not considered appropriate.

3. Questions or advice about errors that may be encountered.

   - Such as "If you see a `scala.MatchError` you should probably add a catch-all `_` case to your `match` expressions."

Language library resources

Unless explicitely stated in the questions, it is not expected that you will use any language library resources in the homeworks.

Possible exceptions to this rule include implementations of datatypes we discuss in this course, such as lists or options/maybes, if they are included in a standard library instead of being builtin.

Basic operations on such types would also be allowed.

- For instance, `head`, `tail`, `append`, etc. on lists would not require explicit permission to be used.

- More complex operations such as sorting procedures would require permission before you used them.

Additionally, the standard *higher-order* operations including `map`, `reduce`, `flatten`, and `filter` are permitted generally, unless the task is to implement such a higher-order operator.

Part 1: Filtering streams [10 points]

Using the custom `Stream` type for this assignment, defined in "a type for this assignment" above, create a method `filter` on `Stream`.

The first argument to `filter` should be a predicate (i.e., a `Boolean`-valued function/method), and the second argument the `Stream` to be filtered.

The result of `filter(p,s)` should be the stream obtained by removing all elements of `s` which do not satisfy `p`; that is, for an element `e` of `s`, if

`p(e)` is `true`, then `e` is in the new stream; otherwise, if `p(e)` is `false`, then `e` is not in the new stream.

Take care!

If you encounter problems in this definition or those below, do review the lecture/tutorial notes; in particular, see the subsection "Take care: make arguments of type `Stream` by name arguments".

Also, note that when working with infinite lists, sometimes it is expected that you will encounter infinite recursion (and hence stack overflows). In particular, think about the result of filtering out *every* element in an infinite stream.

Part 2: Zipping and merging streams [20 points]

Create the following two methods on streams.

1. `zip`, which when given two streams *of equal length*, but potentially with different element types, returns a stream of *pairs* consisting of the elements of those streams.

   - That is, if the two streams were written `a ,a ,a ,…` and `b ,b ,b ,…` , the result of zipping them would be `(a ,b ),(a ,b ),(a ,b )` .

   In the case that the streams are of differing lengths, your method should still not cause an exception. Instead, it should zip together as many elements as possible, but the remaining behaviour is unspecified.

2. `merge`, which when given two streams *with the same type of elements*, but potentially of differing lengths, merges the stream, such that

   - the first element of the resulting stream is the first element of the first stream,
   - the second element of the resulting stream is the first element of the second stream,
   - the third element of the resulting stream is the second element of the first stream,
   - the fourth element of the resulting stream is the second element of the second stream,
   - and so on.

We do not fully specify the behaviour of `merge` if the two streams are of different length, but it should not cause an exception.

Part 3: "Quantifying" over streams [10 points]

Define the two higher order predicates `all` and `exists` on streams, which correspond to the and quantifiers respectively.

- `all(p,s)` returns `false` if there is an element of `s` which does not satisfy the predicate `p`.
- `exists(p,s)` returns `true` if there is an element of `s` which satisfies the predicate `p`.

(Do take note of what behaviour is specified or is not specified here; there is a reason I have stated the behaviour this way.)

Part 4: Tolerant zipping and merging of streams [10 bonus points]

Implement alternative versions of the `zip` and `merge` methods from part 2, called `zipSafe` and `mergeSafe`.

These versions must return a sensible result which contains, in some way, **every element** from **both** streams, even if the conditions for using `zip` and `merge` that were given in part 2 are not met; that is, `zip` must work for streams of differing lengths, and `merge` must work for streams with different element types.

You may make whatever design decisions you feel are necessary to implement these methods. Your solution may be (sometimes subjectively) judged based on

- how close your methods come to the original specified behaviour of `zip` and `merge`,
- whether information (such as type information or elements) is lost by your methods, and
- the overall elegance of solution,
  - in particular whether it fits the *functional* paradigm.

Testing

Unit tests for the requested methods/functions are available here: `./testing/h4/h4t.sc` The contents of the unit test file are also repeated below.

The tests can be run by placing the `h4t.sc` file in the same directory as your `h4.sc` file, and running the following command.

```
amm h4t.sc
```

If you use an implementation other than Ammonite, you may need to adjust the tests to be able to run them on your system.

In particular, those using IntelliJ for writing Scala may need to remove the import statement at the top of that file, or move the tests to their source file (if so, be sure to remove them before submission!)

Automated testing via Docker

The Docker setup and usage scripts are available at the following links. Their contents are also repeated below.

- Dockerfile
- docker-compose.yml
- setup.sh
- run.sh

Place them into your `h4` directory where your `h4.sc` file and the `h4.plt` (linked to above) file exist, then run `setup.sh` and `run.sh`.

You may also refer to the README for this testing setup and those files on the course GitHub repo

Note that the use of the `setup.sh` and `run.sh` scripts assumes that you are in a `bash` like shell; if you are on Windows, and not using WSL or WSL2, you may have to run the commands contained in those scripts manually.

The tests

The contents of the testing script are repeated here.

```
import $file.h4, h4._

// Included in case the student did not define it.
```

```
// But renamed in case they did.
def testTake[A](n: Int, s: => Stream[A]): List[A] = s match {
  case SNil => Nil
  case Cons(a,f) => n match {
    case n if n > 0 => a :: testTake(n-1,f())
    case _ => Nil
    }
  }

/* Given an expected result and a computed result,
   check if they are equal in value.
   If so, return 0. Otherwise, inform the user, and return 1,
   so the number of failures can be counted. */
def test[A](given: A, expected: A, the_test: String) =
  if (!(given equals expected)) {

    ↪  println("+-------------------------------------------------")
    println("| " + the_test + " failed.")
    println("| Expected " + expected + ", got " + given + ".")

    ↪  println("+-------------------------------------------------")
    1
  } else {
    0
  }

// We can define a stream of all natural numbers by
// first defining streams which start at a given integer.
def intsFrom(n: Int): Stream[Int] = Cons(n, _ =>
↪  intsFrom(n+1))
val nats = intsFrom(0)

// Filter test streams
val evenNats = filter((x: Int) => x % 2 == 0, nats)
val natsGtr100 = filter((x: Int) => x > 100, nats)


// The tests are saved as tuples, the pieces of which will be
↪  passed
// to test.
```

```
val tests = List(
  (testTake(10,evenNats),
    List(0,2,4,6,8,10,12,14,16,18),
    "First 10 even nats"),
  (testTake(10,natsGtr100),
    List(101,102,103,104,105,106,107,108,109,110),
    "First 10 nats greater than 100"),
  (testTake(10,zip(evenNats,natsGtr100)),
    List((0,101),(2,102),(4,103),(6,104),

         ↪  (8,105),(10,106),(12,107),(14,108),(16,109),(18,110)),
    "zip test"),
  (testTake(20,merge(evenNats,natsGtr100)),
    List(0,101,2,102,4,103,6,104,8,105,
      10,106,12,107,14,108,16,109,18,110),
    "merge test"),
  (all((x: Int) => x < 100, nats), false, "not all nats less
  ↪  than 100"),
  (exists((x: Int) => x > 100, nats), true, "exists nat
  ↪  greater than 100"),
  )

// Apply test to each element of tests, and sum the return
↪  values.
// This is essentially a for loop.
val failed = tests.foldLeft(0) {
  (failures, next) => next match {
    // Deconstruct the tuple to get its parts
    case (given, expected, the_test) => failures + test(given,
    ↪  expected, the_test)
  }
}

println("+-------------------------------------------------")
println("| " + failed + " tests failed")
println("+-------------------------------------------------")
```

The Docker setup

The contents of the Docker setup files and scripts are repeated here.

```
# This Dockerfile has two required ARGs to determine which
↪  base image
# to use for the JDK and which sbt version to install.

# Define the argument for openjdk version
ARG OPENJDK_TAG=8u232
# Do the packaging based on openjdk
FROM openjdk:${OPENJDK_TAG}

# Set the name of the maintainers
MAINTAINER Habib Ghaffari Hadigheh, Mark Armstrong
↪  <ghaffh1@mcmaster.ca, armstmp@mcmaster.ca>

RUN apt-get update && \
  apt-get install scala -y && \
  apt-get install -y curl && \
  sh -c '(echo "#!/usr/bin/env sh" && \
  curl -L
↪  https://github.com/lihaoyi/Ammonite/releases/download/2.1.1/2.12-
↪  2.1.1) > /usr/local/bin/amm &&
↪  \
  chmod +x /usr/local/bin/amm'

# Set the working directory
WORKDIR /opt/h4

version: '2'
services:
  service:
    build: .
    image: 3mi3_h4_docker_image
    volumes:
      - .:/opt/h4
    container_name: 3mi3_h4_container
    command: ['amm', 'h4t.sc']

docker-compose build --force-rm

# Run the container
docker-compose up --force-recreate
```

```
# Stop the container after finishing the test run
docker-compose stop -t 1

# Instructions for automated testing using Docker

We have already created a `Dockerfile` here which specifies
all the necessary packages, etc., for compiling and running
↪  your code.
You just only need to follow the instructions below to see
the results of unit tests designed to check your
↪  implementation.

## Setup
We use `docker-compose` and its configuration file to build
↪  the image.
Assuming you have `docker` and `docker-compose` installed,
simply execute
```shell script
./setup.sh
```
 to generate the image.

Prepare your code for the running the tests
You only need to place the `h4t.sc` unit test file and
the `run.sh` file in the same directory as your `h4.sc` source
↪ file.

Running the tests
As with the build process, we have already put
the configuration needed for running the test inside
↪ `docker-compose.yml`.
Simply execute
```shell script
./run.sh
```
to run and see the results of the tests.
```