COMP3141 Assignment 2

H.A.R.E.

Zoltan A. Kocsis

University of New South Wales
Term 2, 2022

Marking Total of 20 marks (20% of practical component)

Due Date Tuesday, 2 Aug 2022, 23:59.

Submission Instructions The assignment can be submitted using the '`give`' system. To submit from a CSE terminal, type:

```
$ give cs3141 assignment2 Hare.hs
```

You will be working in a single file, `Hare.hs`. This is the only file you should modify, and the only file you will get to submit. The other two files, `RoverModel.hs` and `RoverInterface.hs` contain useful information, but should not be modified.

Version 18 July 2022

Background

Sending a rover to Mars is a major undertaking even today. 22 years ago, when the joint space program between Tortoise and Hare first proposed the High-Assurance Rover Expedition (H.A.R.E.), it was even harder. A journey to Mars takes over 10 months, and the hardware has to be able to operate for decades without maintenance. Unfortunately, while the hardware is still going strong, the software is starting to show its age.

The original operating system for the H.A.R.E. Mars rover was based on a very old version of TortOS (TortOS 98). It was never intended to be used for more than a few years, and certainly not for the length of time that the H.A.R.E. rover has been in operation. The system is becoming increasingly unstable, and there have been a number of glitches and failures in recent months.

It is now time for an upgrade, and the joint space program has contracted you to help. Unfortunately, there are a two complications. First of all, the OS files have to be sent here via radio. The H.A.R.E. rover is currently located in a canyon, and will have to be repositioned to allow for a better downlink. Your first task will be to write a program that helps the rover find its way out of the canyon, to a more favorable location.

Secondly, installing the new version of TortOS will involve rebooting the rover's computer. The rover has no hard drive: rebooting it will erase the contents of the rover's in-memory file system, causing the rover to lose its current location and all of its mapping data.

By happy coincidence, one of the experiments carried by the H.A.R.E. rover contains a obsolete type of magnetic storage media known as a *floppy disk*. While its capacity is small, this disk can be used to store the contents of the in-memory file system, so that they can be restored after the reboot. Your second task is to write a disk controller program that will transfer the contents of the file system to the floppy disk before rebooting, and restore them afterwards.

With these two programs in hand, the H.A.R.E. rover will be able to upgrade to the latest version of TortOS, and continue its mission for many years to come!

Waypoints

Mars has no GPS satellites to aid in navigation. However, over the course of 20 years, H.A.R.E. managed to map its surroundings with an impressive degree of accuracy. It has built up a database of waypoints and landmarks, and should be able to use them to navigate its way around the Martian surface.

There are many different waypoint types: the only thing they have in common is that they each belong to the following type class, `Wp`.

```
class Eq wp => Wp wp where
  navigableFrom :: wp -> [wp]
```

When given a waypoint `x :: wp`, the `navigableFrom` function returns a list of other waypoints that the rover knows how to reach from the current waypoint `x`.

If the waypoint `x` is navigable from the waypoint `y`, then the waypoint `y` is also navigable from `x`: more formally, the equality

```
(x `elem` navigableFrom y) == (y `elem` navigableFrom x)
```

holds for all waypoints `x`, `y`.

However, two waypoints `x` and `y` may not have a direct, navigable route between them. This happens if `not (x `elem` navigableFrom y)`. If so, it might still be possible to get from from point `x` to point `y` indirectly, through one or more intermediate stops: e.g. one might have to navigate from point `x` to point `z`, and then from point `z` to point `y`.

Paths

The `Path wp` data type, defined below, is a data structure for storing such indirect routes between two waypoints.

```
data Path wp
  = From wp
  | GoTo (Path wp) wp
  deriving (Eq)
```

The data type has two constructors. `From x` is a one-element path: following it just means staying at the waypoint `x`. To follow `GoTo xs x`, the rover will first follow the path `xs` from its starting location to its endpoint, then navigate from there to the waypoint `x`.

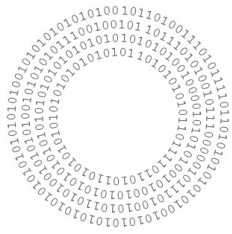
Each element of type `Path` must obey the following well-formedness conditions:

1. A path `Form x` is always well-formed.
2. A path `GoTo xs x` is not well-formed unless `x` is navigable from the endpoint of the path `xs`, and the path `xs` is itself well-formed.
3. A path is not well-formed if it contains the same waypoint more than once.

Floppy Disks

The *floppy disk* that we'll use to store our data is an 8-inch-diameter (200mm) flexible magnetic memory disk, capable of holding a whopping `80 KiB` (`81920 bytes`) of data.

The data is stored in 40 circular closed *tracks* along the disk, as depicted below:

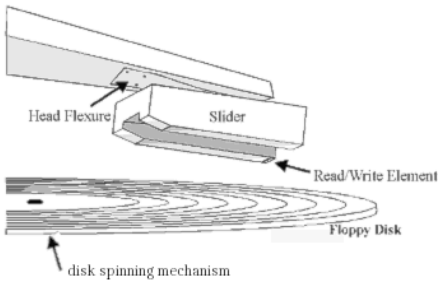


The tracks are numbered from 0 to 39: track 0 is on the outermost edge, while track 39 is on the inside. While the inner tracks are physically shorter than the outer tracks, each track stores the same amount of logical data: `2048 bytes`.

Disk Drive

The disk drive is capable of reading/writing data from/to the floppy disk.

The drive itself consists of a rotary actuator that can rotate (*spin*) the disk, and a read-write element (the *head*) suspended on an arm above the disk, as depicted below:



The head is always positioned above one of the 40 tracks. The head can slide radially: *forward* (toward the center of the disk) or *backward* (toward the edge of the disk).

The head is always above a specific track. When the appropriate command is given (`readTrack`), the disk starts spinning, the head reads the contents of the track underneath it, and the drive returns all `2048 bytes` of data to the system. Similarly, on a `writeTrack d` command, the drive writes the given data onto the track under the head, so that it can be read back in the future.

The head can be aligned to a track, but it cannot be aligned to a particular position on the track, and there's no guarantee that the disk will return to the exact same position after a full spin. All in all, the circular track has no definite end or beginning: reading or writing might start from a random point.

The disk drive is controlled by *monadic* code: any monad that is an instance of the `MonadFloppy` class is able to control the drive.

```
class (Monad m) => (MonadFloppy m) where
  headForward  :: m ()
  headBackward :: m ()
  readTrack :: m [Word8]
  writeTrack :: [Word8] -> m ()
```

The `Word8` type here is an 8-bit unsigned integer (i.e. a byte).

File System

The rover has a primitive in-memory file system, which is organized into *files* and *directories*. The file system is hierarchical: each directory may contain files and other directories inside it, and those directories can in turn contain files and directories, and so on. The files and directories do not have names: each one is identified by a `Word8`, its UID. The UID is locally unique: i.e. the same directory cannot contain two things with the same UID.

You can make the following assumptions about the file system of the rover:

1. The total size of all the files is no more than `16KiB` (`16384 bytes`).
2. Each single file is at most `3072 bytes` long.
3. There are no more than 48 files and directories altogether.

In Haskell, file system hierarchies are represented by the following recursive type:

```
data FSH a
  = File UID a
  | Dir UID [FSH a]
    deriving (Show, Eq)
```

Each file in the hierarchy is annotated with a value of type `a`: this could be used to represent the contents of the file, or some other user-specified data.

Problem 1: Waypoint Navigation (5 marks)

a. Implement a function `wf :: (Wp wp) => Path wp -> Bool` that tests whether a given path is well-formed according to the conditions given in the specification above.

b. Implement a smart constructor `(>:>) :: (Wp wp) => Path wp -> wp -> Maybe (Path wp)` corresponding to the 'GoTo' constructor. The smart constructor should return `Nothing` if adding the given waypoint to the given path would result in a non-well-formed path, and `Just` the constructed path otherwise.

c. Write a function `extendPath :: (Wp wp) => Path wp -> [Path wp]` which returns the list of all possible ways of extending the given `Path` by appending to it a single additional waypoint.

d. Implement a function `findPaths :: (Wp wp) => Path wp -> wp -> [Path wp]` which returns all possible ways of extending the given `Path` (by appending *any* number of additional waypoints) into a path that ends at the given target waypoint.

Efficiency mark: You'll get the efficient implementation mark for this subproblem if your implementation can gracefully handle weird edge cases.

Marking Criteria	
Marks	Description
1	correctly implementing `wf`
1	correctly implementing `>:>`
1	correctly implementing `extendPath`
1	correctly implementing `findPaths`
1	efficiently implementing `findPaths`
5	**Total**

Part 2: Rotation-invariant Coding (5 marks)

The floppy disk drive has no means of tracking the angular position of the spinning magnetic disk. This means that in principle, reading/writing can begin at any position within the track, and the user has no control over where the reading/writing starts from.

For example, if you were to write the bytes `[1,2,3,4]` onto a track that has a capacity of `4 bytes`, the floppy drive may read them back later in any of the following orders: `[1,2,3,4]` or `[2,3,4,1]` or `[3,4,1,2]` or `[4,1,2,3]`.

In this subproblem, you will come up with a data encoding scheme that lets you read your data unambiguously even if it is read back starting from a different byte. The following data type will represent a list of bytes encoded using your scheme:

```
data Encoded = Encoded [Word8] deriving (Show, Eq)
```

a. Implement a function `rotate :: Int -> Encoded -> Encoded` which simulates the effect of the spinning disk by rotating the given list to the left by the given positive integer number of entries. Hint: you don't have to deal with negative inputs.

b. Come up with an encoding scheme which gets around the problem of the spinning disk by remaining decodable under rotations. More formally, implement a pair of functions, `encode` and `decode`, so that if you start with a list of bytes, `encode` them into your encoding scheme, then rotate the encoded list any number of times, then `decode` it, you end up with the list of bytes you originally started with.

Efficiency marks: You'll get two marks if you provide a space-efficient implementation of *encode*. More specifically, you'll get the mark if you're able to encode every input list of length 16 or below into an encoded output list of length 37 or below.

Marking Criteria	
Marks	Description
1	correctly implementing `rotate`
1	implementing `encode` and its inverse`decode`
1	encoded bytes remain decodable after rotations
2	efficiently implementing `encode`
5	**Total**

Part 3: File System Hierarchy (4 marks)

Since upgrading the operating system requires rebooting the rover's computer, and the reboot will erase the contents of the rover's in-memory file system, we will have to transfer the contents of the file system onto the floppy disk.

It has been decided that one track on the floppy disk will be used to store the contents of at most one file, i.e. that one traack will not store multiple files, not even if it would otherrwise have the spare capacity to do so.

However, we might have to store a single file using multiple tracks, since floppy tracks have a capacity of `2048 bytes`, and a single file may take up more space than that.

We will divide each file into a list of consecutive encoded *chunks*, in such a way that each chunk is short enough to be stored in a single track. We will store each chunk in a unique track. Chunks can be represented by the Haskell data type below:

```
data Chunk =
  Chunk TrackNo Encoded deriving (Show, Eq)
```

We will store the mapping which assigns each file to its list of chunks on a separate track, called the *header*. To reassemble the files, we will have to first read and decode the header, then use it to read and decode each of chunk of the files from the disk in order.

a. Write a monadic function `chunks :: [Word8] -> State TrackNo [Chunk]` which, when given the contents of a file (as a list of bytes), divides it into a list of `Chunk`s.

b. To complete the implementation of the `mkHeader` function, write a lawful `Functor` instance for the FSH type.

c. Implement a function `assignTracks :: FSH [Word8] -> Maybe (FSH [Chunk])`, which, if possible, divides all files in the given hierarchy into chunks, and allocates to each chunk a unique track number which will store the contents of the given chunk when the file system is written to disk. Do not allocate track 0 to any chunk, as track 0

will later be used to store the header. Return `Nothing` if the given file system would not fit on tracks 1-39 of a 40-track disk under your encoding.

Marking Criteria	
Marks	Description
1	correctly implementing `chunks`
1	implementing a lawful `Functor FSH` instance
2	`assignTracks` meets its specification
4	**Total**

Part 4: Monadic Disk Control (6 marks)

`Monad`s that are instances of the `MonadFloppy` class allow you to write monadic programs for controlling a floppy disk drive. Apart from the usual monad operations (bind/return), it has four additional operations, with the following effects:

- `headForward` - moves the r/w head forward by 1 track. Has no effect if the head is already aligned with track 40.
- `headBackward` - moves the r/w head backward by 1 track. Has no effect if the head is already aligned with track 0.
- `readTrack` - reads and returns 2048 consecutive bytes from the track currently under the head.
- `writeTrack` - writes its first argument onto the track currently under the head.

You will be writing monadic functions which work with any monad `m` that belongs in the `MonadFloppy` class. An example of such a monad is provided in the file `RoverModel.hs`: this will allow you to test your monadic programs `p :: m ()` by calling `runDiskModel`, similarly to how you would call `runState` for the `State` monad.

You also have an example of a monadic program, `saveHeader`, pre-implemented for you.

You can assume that reading or writing a track will always succeed, and that the actual implementation of the four functions above (the implementation running on the rover) abides by the specification above. Beware: any behavior that the specification leaves unspecified (ill-formed arguments, etc.) is implementation-defined. Your code should not assume a specific implementation, and should behave correctly for any possible monad that belongs to the `MonadFloppy` class.

a. Write a monadic program `headToTrack :: (MonadFloppy m) => TrackNo -> m ()` that positions the r/w head of the disk drive on the track with the given track number. If the given number is larger than 39, position the head on track 39.

b. Implement a program `saveChunk :: (MonadFloppy m) => Chunk -> m ()` which writes the given chunk onto its allocated track of the disk.

c. Implement a program `saveFSH :: (MonadFloppy m) => FSH [Word8] -> m Bool` that attemps to save the given file system onto the disk. It should first attemp to assign tracks to the given input file system hierarchy using 'assignTracks'. If the assignment was unsuccessful, the program should return `False` and not make any changes to the disk. However, if the assignment was successful, the program should write the corresponding header to track 0 of the disk, then write all the assigned chunks onto the appropriate tracks, and finally return `True`.

d. Implement a program `loadFSH :: (MonadFloppy m) => m (Maybe (FSH [Word8]))` that is able to reload a file system saved by `saveFSH` from the disk.

Hint: To load the saved header, you might want to study the `fromBytes` function and its implementation in `RoverInterface.hs`.

Efficiency mark. The final mark is an end-to-end test of all your programs on a much more accurate simulation of the rover. If you're unsuccessful, you'll get a short log about the fate of the rover.

Marking Criteria	
Marks	Description
1	correctly implementing `headToTrack`
1	`saveChunk` passes simple tests
1	`saveFSH` passes simple tests
2	`loadFSH` can reload saved systems
1	end-to-end test passed
6	**Total**

End of Problem Statements.

Overview

This assessment is intended to give you experience writing monadic Haskell code, as well as experience reading, writing and complying with mathematical specifications mimicking simplified versions of real-world problems. It also tests your knowledge of common monads, including `Maybe`, `State`, `[ ]`.

You have more than two weeks to complete the assignment, but beware: a full solution will involve writing approximately 180 lines of Haskell code, and doing quite a bit of thinking. Don't leave it until last minute!

Historical Background. The scenario presented in this assignment is inspired by real world events. Mars Express was the European Space Agency's first mission to the Red Planet. Launched in 2003, the orbiter has already spent almost two decades studying Earth's neighbour and revolutionising our understanding of Mars. In 2022, Mars Express underwent a major remote software upgrade. It included a series of upgrades that improve signal reception and on-board data processing, and greatly increased the amount and quality of science data sent to Earth.

Marking and Testing

All marks for this assignment are awarded based on *automatic marking scripts*. Marks are not awarded subjectively, and are allocated according to the marking criteria presented at the end of each problem statement. The scripts that run when you submit the assignment are similar to (but not identical with) the scripts that will be used to determine your final marks: you are advised to do your own testing, instead of relying solely on the submission test suite.

Barring exceptional circumstances, the marks awarded by the automatic marking script are *final*. For this reason, please make sure that your submission compiles and runs correctly on CSE machines, and that the submission scripts do not report any problems.

Late Submissions

Unless otherwise stated if you wish to submit an assignment late, you may do so, but a late penalty reducing the maximum available mark applies to every late assignment. The maximum available mark is reduced by 5% if the assignment is one day late, 10% if the assignment is two days late, 15% if the assignment is three days late. Assignments that are late 4 days or more will be awarded zero marks.

Extensions

Assignment extensions are only awarded for serious and unforeseeable events. Having the flu for a few days, deleting your assignment by mistake, going on holiday, work commitments, etc do not qualify. Aim to complete your assignments well before the due date in case of last minute illness, and make regular backups of your work.

Plagiarism

All work submitted for assessment must be **entirely your own**. Unacknowledged copying of material, in whole or part, is a serious offence. Before submitting any work you should read and understand the UNSW Plagiarism Policy.

In this course submission of any work derived from that of another person, or solely or jointly written by or with someone else, without clear and explicit acknowledgement, will be severely punished, including with automatic failure and an overall mark of zero for the course. This includes using unreferenced work taken from books, web sites, etc.

Do not share your work with any other person! Allowing another student to copy you work will, at the very least, result in zero for that assessment. If you knowingly provide or show your work on this assignment to another person for any reason,, and work derived from it is subsequently submitted, you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep all your work private. If you are unsure about whether certain activities constitute plagiarism, you should ask us before engaging in them!