Lab Exercises for COMP26020 Part 2: Functional Programming in Haskell

Joe Razavi

February 3, 2023

The deadline for this lab is 6pm on 17/2/2023.

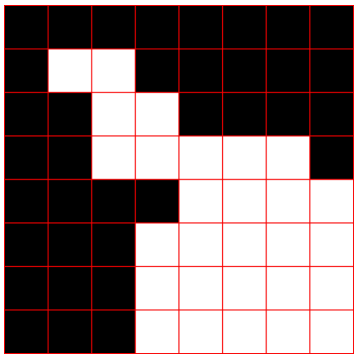
This lab has three exercises, for a total of **ten marks**. The first two exercises together are worth **eight marks**, and I advise all students to focus exclusively on these exercises. Seven marks are given based on automated testing, and one is reserved for human judgement by the marker. These exercises are described in Section 1 below. Section 2 contains submission information and a checklist of tasks for the first two exercises.

If you are certain that your solutions are completely correct you might like to look at Section 3 below, which describes a thought-provoking, open-ended exercise requiring significant creativity, worth **two marks**. It is designed to be extremely difficult, and is not a practical way of gaining marks!

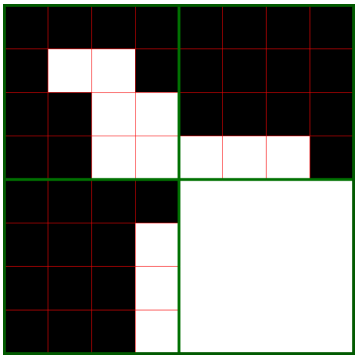
1 Simple Quadtrees

This lab exercise concerns as data structure called a 'quadtree' which can be used to represent an image. There are sophisticated versions of the quadtree data structure, but for the purposes of the lab we will use a very simple version of the idea.

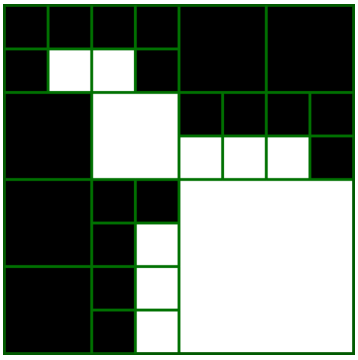
Suppose we want to represent a square, black and white bitmap image which is 2^n by 2^n pixels. The usual way to do this is as a 2^n by 2^n grid of bits, but this can be wasteful if there are large monochrome areas.



In that case, a simple optimization is to think of the image as split into four sub-images of size 2^{n-1} by 2^{n-1} which we will call 'quadrants'. If the sub-image is all one colour, we can represent this by one bit of information.



But if it contains different colours, we can subdivide again, and keep going recursively until we *do* get sub-images which are only one colour. (This definitely happens once we get down to the scale of the original pixels!). We call these single colour sub-images in the final data structure 'cells'.



This lab exercise is about the resulting data structure, the tree of cells. You don't have to care about the details of an original image which such a structure might have come from – for instance you don't need to record the dimensions in pixels of the original image, nor do you need to worry about whether a particular structure is the most efficient way of representing a given image. That means that your data structure is correct if it represents the way the image looks geometrically, ignoring the size. The way you think about the data structure could differ from someone else by rotation, scaling, and even reflection, as well as the details of how you order and organize the various components, and you can both be correct as long as you are each internally self-consistent.

For that reason, if you are working out what the exercises mean with a friend, or asking something on the forum, you should describe everything in terms of pictures, or describe quadtrees using the four functions in Exercise 1 below, so that you don't accidentally discuss the details of your data structure.

1.1 Exercise 1: (3 marks) representing quadtrees

For this exercise, you should define an Algebraic Data Type (that means a custom type defined by the `data` keyword) to model quadtrees in the sense described above. Do this in whatever way you like (as long as you use an Algebraic Data Type), but provide four functions with the following properties:

- A function `allBlack` which takes an `Int` number n and returns your representation of a single cell which is all black. The argument n represents the image 'size', but since all-black images of any size look the same, you can ignore this argument! (See notes below...)
- A function `allWhite` which takes an `Int` number n, as above, and returns your representation of a single cell which is all white.
- A function `clockwise` which takes four quadtrees and returns the quadtree whose four subtrees are the given inputs, arranged in a clockwise order.
- A function `anticlockwise` which takes four quadtrees and returns the quadtree whose four subtrees are the inputs, arranged in an anticlockwise order.

Note the following:

- For `allBlack` and `allWhite` the 'size' argument can be ignored, but for some ways of modelling it might be useful. Neither using the argument nor ignoring it is the 'best' approach: there are a huge number of different correct solutions! You can assume that any testing data come from real bitmaps (square bitmaps whose width is a power of 2), and the 'size' arguments tell us how many pixels wide each cell is in the originating bitmap.
- A clockwise ordering means that in the tree `clockwise a b c d`, the subtree `b` is located in the quadrant next to `a` which is reached by moving clockwise, `c` is in the quadrant reached by moving clockwise from `b`, `d` is in the quadrant reached by moving clockwise from `c`, and `a` is in the quadrant reached by moving clockwise from `d`.
- For `clockwise` and `anticlockwise` it doesn't matter how subtrees are stored or ordered internally, or which quadrant comes 'first' – a correct solution is still correct if we rotate, reflect, or scale everything and all tests involved in marking will respect this. However, the choices you make should be consistent: the clockwise and anticockwise orderings must be opposite to each other, and all uses of `clockwise` and `anticlockwise` in your solution should make the *same* choice about which quadrant the first argument goes in!

You **must** use at least one Algebraic Data Type in your model, but you may use several. For each Algebraic Data Type, you must add the expression `deriving (Eq, Show)` to the end of the line which defines the datatype.

For example, below is an Algebraic Data Type representing a list of `Int` values

```
data MyList = Elist |
  Cons Int MyList
```

If I used such a data structure in my solution, I would append the expression above to the end of the definition, to obtain

```
data MyList = Elist |
  Cons Int MyList deriving (Eq, Show)
```

Make sure you have done this for all the Algebraic Data Types you have defined.

For now we treat this as a 'magic incantation' which lets Haskell know we want to be able to print values of our datatype and compare them for equality. What is really going on in this expression will be covered in the videos in the last week of the Haskell part.

Note that the four functions completely specify how the quadtree is split up into cells.: it is best not to optimize when given an input which is not efficiently encoded. You won't actually lose marks for this, but it makes exercise 2 harder if you optimize the tree structure!

Marking Exercise 1

This exercise is has a total of **three marks** available. The marks will be assigned based on testing on quadtrees of different sizes and complexities.
The tests will consist of checking consistency properties which we expect to hold. For example, we expect

```
clockwise (allBlack 1) (allBlack 1) (allWhite 1) (allWhite 1) ==
  anticlockwise (allBlack 1) (allWhite 1) (allWhite 1) (allBlack 1)
```

The tests will also involve checking *inequalities* such as

```
clockwise (allBlack 1) (allBlack 1) (allWhite 1) (allWhite 1) /=
  anticlockwise (allBlack 1) (allBlack 1) (allWhite 1) (allBlack 1)
```

Otherwise you could represent all trees with a single value! Note however that they do not check anything which depends on the size of the image, so for instance they never check whether `allBlack 128 == allBlack 2`, because geometrically these look the same, so you are free to represent them as the same or different, whichever works for your data structure. The tests only check equalities and inequalities which *must* hold for all correct representations.

You solution will receive:

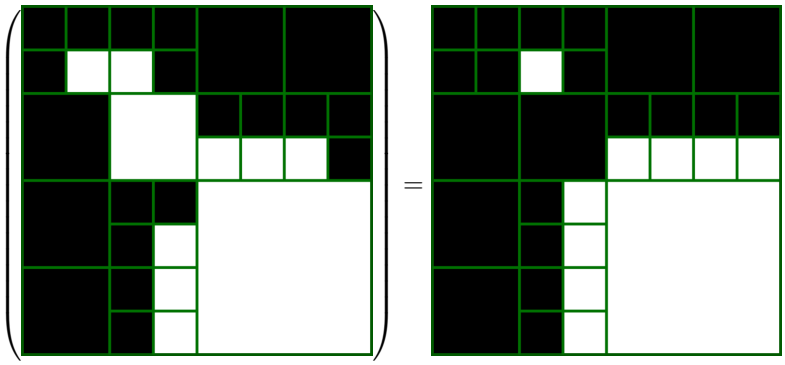
- **One mark** for passing the tests on quadtrees described at most one use of `clockwise` or `anticlockwise` (so they either have 1 cell or 4 cells),
- **One mark** for passing the tests on quadtrees which represent 4 by 4 images (they can have up to 16 cells, but may have fewer if they have an interesting structure),
- **One mark** for passing the tests on all quadtrees,

for a total of three marks. The quadtrees used for testing are no larger than required to represent a 2^{10} by 2^{10} image. You need only consider square images whose dimensions are powers of 2. Your solution must use at least one Algebraic Data Type to qualify for any of the marks above.

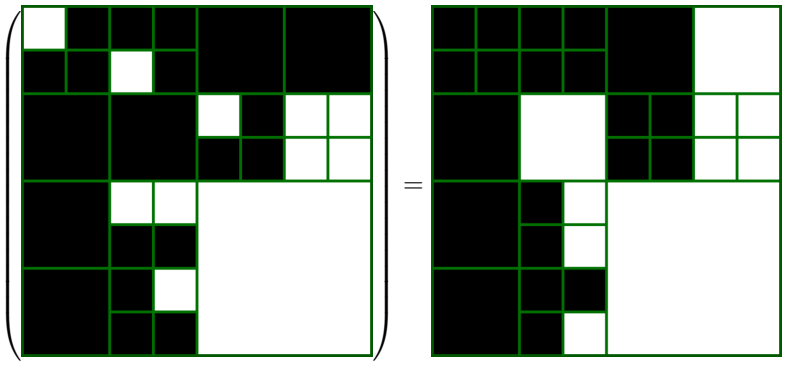
1.2 Exercise 2: (4 marks) A crude 'blurring' operation

For this exercise, you should define a function `blur` which takes a quadtree as input and returns a quadtree as output. It should not change the structure of the quadtree[1], but it should change the data representing the black and white colours.
To find the colour of a cell in the output `blur q` we look at all of its 'neighbours': other cells which touch it along an edge (or part of an edge), not just a corner. The colour of a cell in `blur q` should be the opposite of the colour of that cell in the input `q` if and only if more than half of its neighbours have the opposite colour in `q`. E.g. if a cell is black in the input `q` then it should be white in the output `blur q` if and only if in `q` more of its neighbours are white cells than black. You can think of such a function as an extremely crude approximation to a blurring operation, although it is not practical to use it for that purpose!

For example

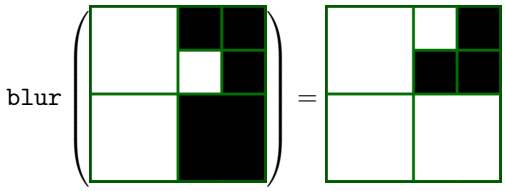
`blur`  =

and

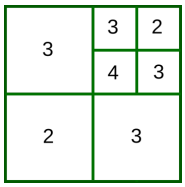
`blur`  =

[1] If your implementations of `clockwise` and `anticlockwise` perform any optimisations, your implementation of `blur` should perform matching optimisations to the *result* of the operation described; most students will not need to worry about this as there are no marks for optimizing `clockwise` or `anticlockwise`.

Note that cells at the border usually have fewer neighbours, so the definition behaves quite strangely there. For example



which shows that in many cases the approximation to 'blurring' is very bad indeed! The image below shows how many neighbours each cell of the quadtrees above has. Note that a cell is not one of its own neighbours.



Coming up with a solution all at once is hard! For that reason, the mark scheme below gives most of the marks for solving special cases. It may be best to try the special cases first if you can't see how to solve the whole problem straight away. To define the special cases, let us call a quadtree **striped** if it satisfies the following recursive specification:

- **Base case:** For all `Int` values `z` (satisfying the conditions from Exercise 1), the quadtrees `allBlack z` and `allWhite z` are striped quadtrees;
- **Step case:** If `q1` and `q2` are striped quadtrees, then `clockwise q1 q1 q2 q2` is a striped quadtree.

Try drawing some pictures of striped quadtrees by building up from the base case. Notice how `q1` and `q2` are repeated in the step case – this repetition means that we already know the values of a cells neighbours in one dimension. (Though we do still have to worry about whether is *has* any neighbours in the other dimension, or if it is on the border.)

Marking

This exercise is has a total of **four marks** available. The marks will be assigned based on testing on quadtrees of different sizes and complexities; more detail will be given in the final version of the lab instructions.
The tests will consist of checking properties which we expect to hold. For example,

```
blur (clockwise (allWhite 2)
                (clockwise (allBlack 1) (allBlack 1)
                           (allBlack 1) (allWhite 1))
                (allBlack 2) (allWhite 2)) ==
 clockwise (allWhite 2)
           (clockwise (allWhite 1) (allBlack 1)
                      (allBlack 1) (allBlack 1))
           (allWhite 2) (allWhite 2)
```

You solution will receive:

- **One mark** for implementing a function `blur` of the correct type which does not 'go wrong' as long as its argument is a suitable quadtree (one which could come from a real image),
- **One mark** for passing the tests on striped quadtrees which represent 1 by 1, 2 by 2, or 4 by 4 images (so have at most 16 cells),
- **One mark** for passing the tests on all striped quadtrees, and
- **One mark** for passing the tests on all quadtrees,

for a total of four marks. The maximum size for test inputs is the same as for Exercise 1.

One additional mark is available for the first two exercises according to the marker's judgement. This mark will be given for clearly commented code which explains your solution or where you got stuck. But it may also be given for other good work towards solving any of the above if the marker feels, according to their judgement, that it is not reflected fairly in the automated mark.

2 Submission

To submit the exercises above, clone the git repository `26020-lab3-s-haskell_<your username>` present in the department's GitLab.

In that directory, save your submission as `submission.hs` and make sure you have done `git add submission.hs`. Remove any definition of `main` from `submission.hs`.[2]

A testing script is provided called `check_submission.sh` (note 'sh' not 'hs'!). Running this file checks that your submission will work with the automated marking script. Note that it creates/overwrites a file called `check_submission_temp_file.hs` by concatenating submission.hs and tester.hs. It does not remove this file after running, so you can inspect it if anything went wrong. This script checks that your solution is in the right format for the automated tests (e.g. that you have used the right function names, added the `deriving (Eq, Show)` incantation where necessary, and have remembered to remove any definition of `main`) but **it does not test your submission well!** Come up with your own test examples, and *reason* about whether your code is correct for all inputs!

You might have to make `check_submission.sh` executable by running `chmod u+x check_submission.sh`.

Once `check_submission.sh` tells you that all its checks have passed, double check that you have added `submission.hs` and push the files on the master branch.

If you decide to try the creative exercise below and think you have succeeded or very nearly succeeded, add your work to the repo as `creative.hs`. Students are strongly encouraged to focus on the exercises above.

Once you are confident that your solution is correct (i.e. after doing more testing than just running the format checking script!), push your final version on the master branch and create a tag named `lab3-submission` to indicate that the submission is ready to be marked.

In most cases marking will be done without any further input needed from you once you have submitted, but in some cases we may need you to explain your solution face-to-face in order to complete marking.

[2] Submissions called 'sumission.hs' will also be accepted due to a typo in a previous version of this exercise, however please try to ensure the correct name as this may delay the marking process.

Talking to each other and using the internet

When talking to other students about the coursework, keep in mind some "do"s and "don't"s:

- **Do** discuss what the exercise means, e.g. what the function `blur` does to examples.
- **Do** discuss example quadtrees in terms of pictures or geometry which doesn't change when rotated, translated, or scaled.
- **Do** discuss example quadtrees in terms of the four functions `allBlack` , `allWhite` , `clockwise` and `anticlockwise`.
- **Do** discuss general Haskell questions, e.g. how to write an Algebraic Data Type using `data`, how `Maybe` or lists work, how to bracket when defining recusive functions, etc.

But:

- **Don't** discuss quadtrees using terminology not in this lab script – that probably contains ideas about how your solution works!
- **Don't** discuss how to define a data structure for quadtrees (inlcuding for instance where your solution puts which quadrant of the pictures)
- **Don't** discuss how to define `blur` or similar operations in terms of recursive functions
- **Don't** show anyone your solution to get help with Haskell syntax – instead try to describe the problem in general or reproduce it in an unrelated example

Talking to others is an important way to learn a subject, and I encourage you to discuss Haskell in general. When discussing the lab exercises with other students, stick to the rules above.

Similarly, when using the internet to help with this coursework

- **Do** use non-interactive resources like tutorials and documentation. I recommend
  - `https://en.wikibooks.org/wiki/Haskell/Other_data_structures`
  - `https://www.haskell.org/tutorial/goodies.html`
- **Do** ask general Haskell questions (not based on the lab exercises!) in forums, chatrooms etc.

But:

- **Don't** discuss anything derived from the lab exercises on interactive services like forums, chatrooms, Stack Exchange, etc.
- **Don't** upload the lab script or anything derived from lab work to file-sharing websites, document-sharing websites, or notes-sharing websites.

Checklist for Exercises 1 and 2

Exercise	Task	Marks	Done?
1	Create a data type using `data` to model quadtrees		
1	Ensure any data type declarations you use end with `deriving (Eq,Show)`		
1	Write the `allBlack` function		
1	Write the `allWhite` function		
1	Write the `clockwise` function		
1	Write the `anticlockwise` function		
1	Ensure the functions work for quadtrees with between 1 and 4 cells	**1 mark**	
1	Ensure the functions work for quadtrees with at most 16 cells	**1 mark**	
1	Ensure the functions work for all quadtrees	**1 mark**	
2	Write a function `blur` from quadtrees to quadtrees (which does anything!)		
2	Ensure your function does not 'go wrong' for any well-defined input	**1 mark**	
2	Make the `blur` function work correctly for striped quadtrees with at most 16 cells	**1 mark**	
2	Make the `blur` function work correctly for all striped quadtrees	**1 mark**	
2	Make the `blur` function work for all quadtrees	**1 mark**	
all	Document your code clearly with comments	**1 mark**	
all	Thoroughly test and reason about your solutions		
all	Test your submission using your own test data		
all	Remove any definition of `main` from your submission		
all	Run `check_submission.sh` and make sure there are no errors		
all	Make sure you have added `submission.hs`, pushed, and tagged your submission correctly		

3 An open-ended exercise

The exercise described in this section is designed to be extremely difficult: no amount of time spent on it is guaranteed to result in gaining marks, and it is best attempted only if the thinking in itself is sufficient motivation. I have told the TAs that they do not need to prepare to support these exercises, so you may need to ask me directly about any questions you have!

For this exercise, we assume that quadtrees do *not* record any size information about their cells. If your data structure for Exercises 1 and 2 records size information, make a different Algebraic Data Type for this exercise which does not do so, and provide implementations for the four functions from Exericse 1, except that `allBlack` and `allWhite` should now take *no arguments*. Note that you should do this exercise in a separate file, so that you don't modify anything to do with your existing solutions!
This means we can now write infinite quadtrees, such as
`let q = clockwise allWhite allBlack allWhite q in q`
which form the basis of the exercise.

First, we define a notion of approximation for quadtrees for each natural number n which we call the **n^{th} coarse work approximation**. This is specified by the following equalities:

- `coarsework 0 q == allWhite`
- `coarsework n allWhite == allWhite`
- `coarsework n allBlack == allBlack`
- `coarsework (n+1) (clockwise a b c d) == clockwise (coarsework n a) (coarsework n b) (coarsework n c) (coarsework n d)`

A value `q` of the quadtree type is called **ergodomestic** if for all natural numbers `n` the expression `coarsework n q` does not 'go wrong' in the sense we have been using that phrase heretofore.
A function which takes a quadtree as input and outputs a `Bool` is called a **fair exercise** if and only if when given an ergodomestic quadtree, it evaluates to either `True` or `False` in finite time.

Your task is to define a function `my_solution` which takes as input a function from quadtrees to `Bool` and outputs a quadtree, with the property that for all fair exercises `f`, `my_solution` evaluates in finite time to a quadtree (which may be finite or infinite), and we have

- `f (my_solution f) == True` if there exists a value of the quadtree type `q` such that `f (q) == True` , and
- `f (my_solution f) == False` if there is no such value.

A correct, well-documented solution is worth **two marks**.