CAB402 Assignment 1

Comparative Programming Paradigms

Individual Assignment

Due: 23rd April 2018

Worth: 30%

Overview

This project is designed to give you experience developing a non-trivial software application using the *functional programming paradigm* and to *compare* the experience to the more typical *imperative/object-oriented programming paradigm*. You will create *three* versions of the same application, each programmed in a different manner:

1. The first version will be implemented using F# and is designed to illustrate the benefits of a pure functional programming approach. This version must not make use of mutable state in any way (i.e. you should not use the mutable keyword, nor any mutable data types such as arrays or dictionaries). Its goal is to be clear and simple; to showcase principles and best practice of functional programming; efficiency should not be a concern.
2. The second version will also be implemented using F#, but may make use of mutable state in order to make the implementation more efficient. The functional style of programming should still be maintained, with mutable state only used where it achieves a necessary and significant performance gain. Other less performance critical part of the code should be kept clear and simple and maintain a pure functional style of programming. The mutable parts of the code should be isolated and separated from the pure functional parts, so that the benefits of the functional approach are maintained for the majority of the code.
3. The third version will be implemented using C# and is designed to showcase the principles and practices of object-oriented programming. This includes creating classes to represent distinct entities and concepts present in the application domain. Interfaces and inheritance should be used according to best OO design practice, along with abstraction and data hiding using private fields.

Code Quality

In all cases, identifier names (for classes, functions, methods, parameters, variables) should be very carefully chosen to clearly convey their meaning. i.e. don't use short cryptic identifier names. All function and method implementations should be kept short and simple and indented consistently. The *correctness* of code can be assessed by testing. The *quality* of code is accessed via the degree to which it can be comprehended by someone who didn't write the code. If we start by just looking at a function or method, based purely on its name and parameters, the reader should already have a very good idea of what the function will do. Then when we look at the implementation of a method or function, it should be immediately obvious what it is doing and it should be easy for the reader to convince themselves that the implementation is correct. The code should follow the principle of "*don't make me think*" - it shouldn't force the reader to think or work too hard to understand what is being done. In some cases comments will be required to achieve this, but in most cases, well written code will be completely comprehensible without the need for comments that merely paraphrase what the code is clearly doing. Basically, you want the code to *read like a book*, and just like a book you should use blank lines to break code up into paragraphs. Complex functions and methods should be implemented using other *helper* functions or methods, so that all functions and methods are kept short and coherent – each function or method should *do just one thing*.

Functional Requirements and Unit Tests

All three implementations should create the same output/results (given the same input). And all must follow the basic segmentation algorithm as described below.

Unit Tests

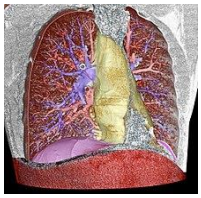
A skeleton solution and set of unit tests have been provided for the first F# version. To obtain full marks, your solution must pass all 17281 unit tests. You should not modify any of the function headers or implementations provided in the skeleton. You should provide implementations for each of the functions marked as not implemented. To do so, you may create and make use of additional private functions.

The second F# implementation does not need to be implemented using the functions provided in the skeleton solution and so does not need to pass any of the provided unit tests. Similarly, the structure of the C# version does not need to mirror the skeleton solution nor pass any of the provided unit tests.

Image Segmentation

From Wikipedia:
Image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as super-pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyse. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

Image segmentation is used in a variety of fields, including for processing medical diagnostic images to help differentiate and outline organs and structures:



This particular project is inspired by a QUT research project that involves analysing periodic satellite images taken over the state of Queensland designed to automatically detect illegal land clearing. The first step of this analysis is to segment each satellite image to identify objects such as roads, fields, rivers, houses, forests, etc. The following is an example of such a segmentation:

The Segmentation Algorithm

The segmentation algorithm that we will implement is a simplification of a more complex algorithm implemented by a commercial image analysis software application called e-Cognition. The segmentation algorithm belongs to a family of segmentation algorithms referred to as "*region growing*" algorithms. In our case, every pixel in the image is initially treated as a separate region/segment. We then try to *merge* individual segments with one of their direct *neighbours* to create larger segments. The new larger segment is referred to as the *parent* segment in relation to the two *child* segments from which it was created. To decide if two neighbouring segments should be merged, a test is performed. Our goal is to create segments that are relatively "*homogeneous*", i.e. the pixels within a segment will be of a *similar* colour compared to pixels in neighbouring segments. Our test is therefore based on the statistical *standard deviation* of the colours of the pixels within the individual segments compared to what the standard deviation would be of the resulting segment if we decided to merge. Each digital pixel colour is represented via a small number of colour *bands*. In the satellite data that we are processing, each colour consists of three 8-bit bands corresponding to *red*, *green* and *blue*. Other satellite images may however have more than three bands, as they may include non-visible infrared and near-infrared bands. So, your code should be written in a general manner that doesn't assume that there will only be three colour bands. Since each pixel colour is made up of a number of separate bands, our test computes the standard deviation of the segments for each band separately and then sums the standard deviation of all bands. The "*cost*" of merging two segments is the standard deviation of the combined segment *minus* the standard deviation of the two individual segments, but with each standard deviation *weighted* by the number of pixels in that segment. Two segments will be merged if their merge cost is less than a constant *threshold*, which is an input parameter to the algorithm. We continue to try to merge neighbouring segments until not further mergers are allowed. If the threshold parameter is increased, then more segments will be merged resulting in fewer final segments.

The order in which we try to merge neighbouring segments is also important. If for example we started in the top left corner of the image and work our way from left to right, we would tend to get very large segments growing in the top left corner, while segments in the remainder of the image would remain tiny. In order to give all segments a chance to grow evenly, we instead try to merge segments in a special order that traverses all points in a more uniformly fair manner that doesn't favour any particular section of the image. Rather than iterating over all segments currently in the image, we instead iterate over all pixels in the image and for each pixel we try to grow the segment that the pixel belongs to. The pixels are traversed in an order determined by a *dither* matrix (https://en.wikipedia.org/wiki/Ordered_dithering). The ordering algorithm is provided for you in the `DitherModule` of the skeleton solution.

When we try to grow a segment we consider each of its neighbouring segments. When determining the neighbours of a segment, two segments are regarded as neighbours if one of the pixels of the first segment is immediately above, below or to the left or right of a pixel in the other segment. In other words, we don't consider pixels to be neighbours if they are only diagonally adjacent. We compute the merge cost of each of the segment's neighbours and select the neighbour(s) with the lowest merge cost. Note that there may be multiple equally best neighbours. For each of those best neighbours, we seek to determine if they are *mutually* best neighbours. For example, if segment B is one of the best neighbours of segment A, then we require also that segment A is one of the best neighbours of segment B. If we can find such a pair of neighbouring segments then we merge them (provided their merge cost is below the threshold). Otherwise we use a technique called *gradient descent*, where we choose one of the current segment's best neighbours and instead try to grow it (recursively), using the same algorithm as outlined above. Fortunately, this recursive gradient descent is guaranteed to terminate.

We iterate over all of the pixels in the image. If during that process any segments were merged, then we iterate over the entire image again. This process continues until no further merging is possible.

F# Data Structures

The `SegmentModule` of the skeleton solution defines a data type for representing segments. There are two kinds of segments: the initial segments that correspond to individual pixels and parent segments that result from the merging of smaller segments. Note that these segments are immutable – once you create a segment, none of its fields will ever change value. To represent a segmentation, the `SegmentationModule` of the skeleton solution defines a data type called `Segmentation` that maps `Segments` to their `Parent` `Segments`. Initially we have only `Pixel` segments, so there are no parent/child relationships between segments. This is represented by `Map.empty`. The F# `Map` type is similar to the `System.Collections.Generic.Dictionary` DotNet type in that they both are generic types that allow you to efficiently lookup a key to retrieve its corresponding value. The difference is that the F# `Map` type is immutable – once you create a `Map`, it can't be updated. The F# `Map` type supports a `Map.add` function, but this function doesn't add a key, value pair to an existing `Map`, it creates a new `Map` that is identical to some previous `Map`, except for the fact that the given key is now mapped to the given value. So, the F# functional segmentation algorithm will start out with an empty `Map`, and as segments are merged, the new parent/child relationships will be captured in new `Maps` that will be created.

The above data structures must be used for the pure version of the F# implementation. For the second impure F# version, you are <u>not</u> required to use any of the types or functions provided in the skeleton solution.

Value Semantics

It is important to understand that all of these F# Segments and Segmentations (Maps) are just *values*, not *objects*. To understand the difference, consider the following C# snippet:

```
var a = new Person(“fred”);
var b = new Person(“fred”);
bool same = (a == b)
```

In this example, `same` would be `false`. Even though both `Person` objects contain the same values, they are considered different because they are separate objects located in different positions in memory. When we compare `a` and `b`, we are really just checking if the reference `a` points to the same place in memory as reference `b` (effectively we are just comparing integer pointer values). This is referred to as *reference semantics*.

Now consider the following F# snippet:

```
let a = Pixel ((0, 0), [52,88,43])
let b = Pixel ((0, 0), [52,88,43])
let same = (a = b)
```

In this example, `same` would be `true`. Two segments are equal if and only if they have the same value for each field. The same applies for Maps (Segmentations); two Maps are equal if they contain the same key/value pairs. This is referred to as *value semantics*.

Your second impure/stateful F# implementation can use any of the DotNet data types, including Arrays and Dictionaries that use reference semantics to increase efficiency.

What to Submit

I. Visual Studio 2017 (version 15.5) solution file should contain 4 projects:

1. Visual F# Console App (.NET Core) [pure]
2. Visual F# Console App (.NET Core) [impure]
3. Visual F# Unit Test Project (.NET Core)
4. Visual C# Console App (.NET Core)

Entire folder structure including solution, 4 project folders, project files and source and test code should be combined into a compressed ZIP file (not tar, not rar, not 7z, etc).

II. A 1 to 2 page PDF report discussing your experience and comparing the three approaches. Your comparison should include an evaluation of each of the following:

(i) the efficiency and effectiveness of the three approaches
(ii) which of the approaches was easier to program,
(iii) which code is easier to read, understand and maintain,
(iv) which of the resulting programs is more concise, and
(v) which produces more efficient code

Where possible, you should support your arguments with quantitative data.