CAB402 Programming Paradigms

Assignment 1

Due Date: 26th April 2019

Weight: 30%

Introduction

The purpose of CAB402 Assignment 1 was to develop a Tic Tac Toe application using pure, imperative, and object-oriented programming skills in F# and C#. This report will discuss the strengths and weaknesses of each paradigm explored in the Tic Tac Toe application and will provide insight into both the practical and conceptual/theoretical issues.

Purely Functional Programming

Purely functional programming usually refers to a programming paradigm that treats all computation as the evaluation of mathematical functions. It is also often defined by the restriction of mutable data types and change in computer state. This ensures that all functions within the functional paradigm will only depend on their arguments regardless of any global or local state.

Purely functional programming has both theoretical benefits and disadvantages including the following:

Advantages	Disadvantages
• Has a higher level of abstraction and more compressed code by hiding a large number of routine operations • Programs are written at a higher level and are therefore more predictable and easier to comprehend • Function signatures are more meaningful as there is no change in hidden states and side effects • Much easier to test and debug as it does not alter any hidden states or have side effects • Parallel programming is much easier and improves performance • Encourages reusability	• Can be more difficult to combine pure functions into a complete application • The immutability of the paradigm requires another set of mathematics and problem-solving skills • Using only immutable values and recursion can lead to performance problems such as RAM use and speed, especially in ill-designed code

Imperative Programming

Opposed to pure functional programming, imperative programming is a programming paradigm that uses statements to change a program's state. Computations are performed through an ordered sequence of steps, in which these states are referred to or mutated. Imperative programming has both benefits and disadvantages as listed below:

Advantages	Disadvantages
• Easy to reason about the source code in terms of what a single core processor would do with it • More time efficient than a fully purely functional program • More space efficient due to the mutability of the program state • High degree of familiarity as many use imperative programming in workplaces and educational institutions • Most languages are imperative and only supports the imperative paradigm	• Lacks the ability to abstract a problem • Can be complex to understand or prove the semantics of a program because of referential transparency does not hold (due to side effects) • Side effects and changes in program state often cause testing and debugging to be harder • Order is crucial which doesn't always suit itself to problems

Object-Oriented Programming

Object-Oriented Programming (OOP) is a paradigm which defines "real-world" objects as separate entities having their own state. Each entity is often modified only by built-in functions and procedures called methods. OOP is an extension of imperative programming. The advantages and disadvantages are as below:

Advantages	Disadvantages
• Easy-to-understand code due to code blocks and the nature of OOP (modelled by real-world concepts) • Provides the abstraction that the imperative paradigm lacks without becoming too complex to understand • Offers reusability of classes and thus, enables faster development • Time efficiency is possible via parallel development of classes • Marginally easier to test, manage, and maintain than imperative programming alone • Secured development technique since data is encapsulated and cannot be accessed via external functions	• Typically involves more lines of code from OOP and simply overhead, resulting in a larger program size • Can be slower due to the number of instructions required to be executed

Purely Functional Programming Experience (F#)

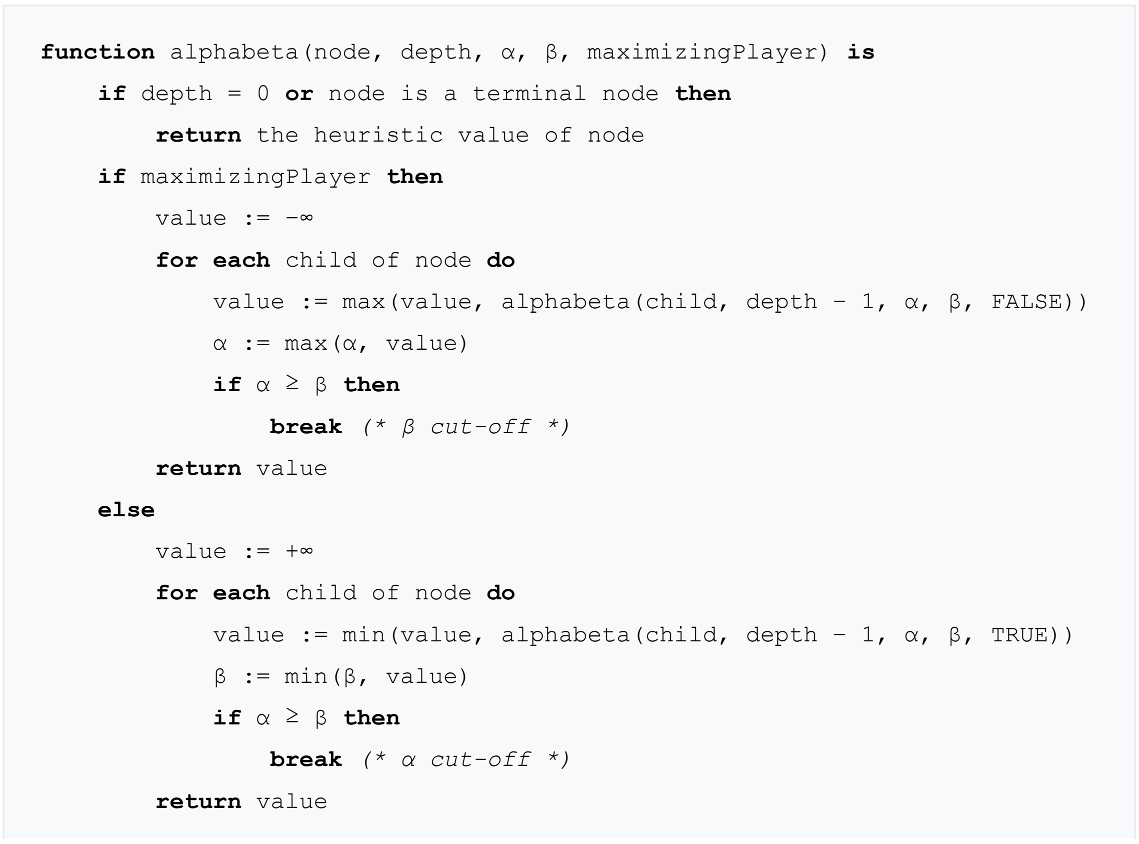
To complete the Tic Tac Toe application, I had first implemented GameTheory.fs and TicTacToePure.fs in pure F#. As purely functional programming was a new experience, it proved difficult to learn to avoid mutable states and to improve knowledge in recursion.

For example, it may have been much easier to allow a *GameState* (refer to *TicTacToeModel.GameState*) to have a mutable *Turn* and *Board*. Doing so would not require a new *GameState* state to be produced each time a move is applied, or a turn is changed. The *GameTheory.MiniMaxGenerator.MiniMax* function was also difficult to implement purely, as *value* in the pseudocode below was a mutable state. This was overcome by pre-generating a List of *Tuples* containing *GameStates* with the *Moves* applied. This was then pipelined to produce a list of recursive *MiniMax* results and its corresponding *Move* in tuple form (refer to *GameTheory.MiniMaxGenerator.MiniMax*).

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

Figure 1: MiniMax Algorithm Pseudocode

Alpha-Beta Pruning in *GameTheory.MiniMaxWithAlphaBetaPruningGenerator.MiniMax* was another challenge as F# did not allow a *break* in *for* loops. The *foreach* loop in figure 2 was transformed into a recursive function where the *alpha* >= *beta* comparison was the base case. As achieved in *GameTheory.MiniMaxGenerator.MiniMax*, a list of *Tuples* containing a *Move* and its applied *GameState* was also used in Alpha-Beta Pruning to avoid mutability in the function.



```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

Figure 2: Alpha-Beta Pruning Algorithm Pseudocode

Impure Programming Experience (F#)

Impure programming in F# was much easier due to the lack of restriction in the mutability of states. For example, *GameState* contained a mutable *Turn* and *Board* so that a new *GameState* did not need to be produced every time a *Move* was applied. A 2D array was also used to represent the *Board* in *GameState*, instead of a *List* of *Lists*. This was done so to allow mutability of the *Board* as *Lists* are considered pure.

The *MiniMax* function with Alpha-Beta Pruning used for impure programming was also changed to contain mutable values. For example, the new alpha and beta values which were produced in *MiniMax* were kept mutable as they required a change in state multiple times (refer to figure 2).

Object-Oriented Programming Experience (C#)

The object-oriented programming (OOP) implementation in C# closely resembled the impure F# implementation as they both were imperative solutions. However, the *Move* and *GameState* were represented as objects and the *Player* was represented by an enum. As the Alpha-Beta Pruning of the OOP implementation was not completed, I unfortunately cannot further discuss implementation changes in the *MiniMax* function. However, according to the pseudocode in figure 2, the *foreach* loop requires a *break*. While a *break* was impossible in F#, C# allows for the use of *breaks* and would not require a recursive function to find the maximum and minimum values for alpha and beta.

Efficiency and Effectiveness of Each Programming Paradigm

Efficiency of Alpha-Beta Pruning

While the *MiniMax* function had produced correct results in the purely functional implementation, the execution time of the function was 19,000 milliseconds during the testing for *TestFindBest()* in Test.cs. Alpha-Beta Pruning was therefore used for the *MiniMax* function which allowed greater efficiency in all the programming paradigms explored. The graph below represents the difference in execution time for the *MiniMax* function with and without Alpha-Beta Pruning in F#.

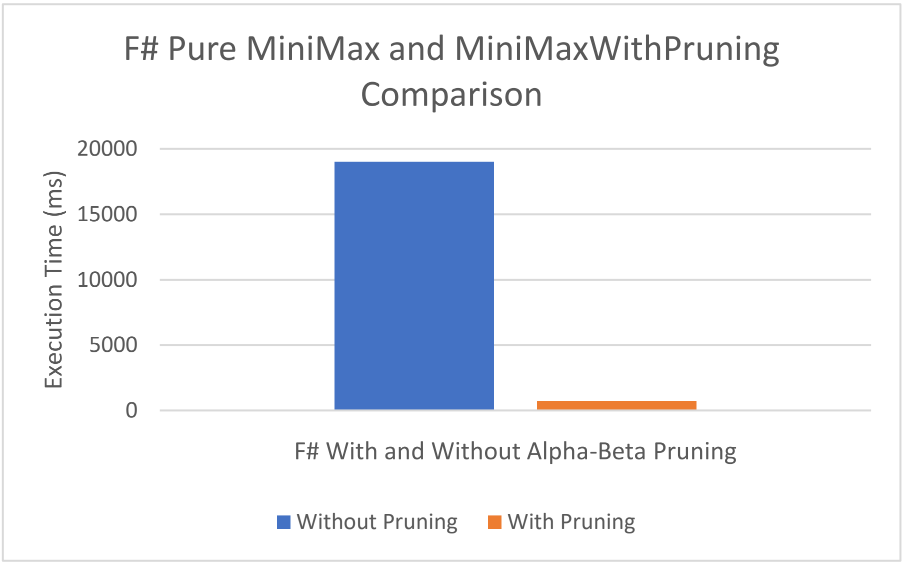


Figure 3: Comparison of Execution Time Between MiniMax With and Without Pruning

As shown above, the difference in efficiency for the *MiniMax* function with and without Alpha-Beta Pruning is significant. The pruning results in a 18,256 millisecond difference (refer to appendix 2).

Efficiency of Each Programming Paradigm

It was hypothesized that a purely functional implementation would have the longest execution time due to the restrictions to mutable data types. As purity restricts the ability to make destructive updates, the creation of a new state was required when making an updated copy of any state. Thus, it can be said that purely functional programming is also more memory-intensive than imperative programming.

A comparison between the execution time of the pure and impure implementation of Tic Tac Toe can be seen as below,

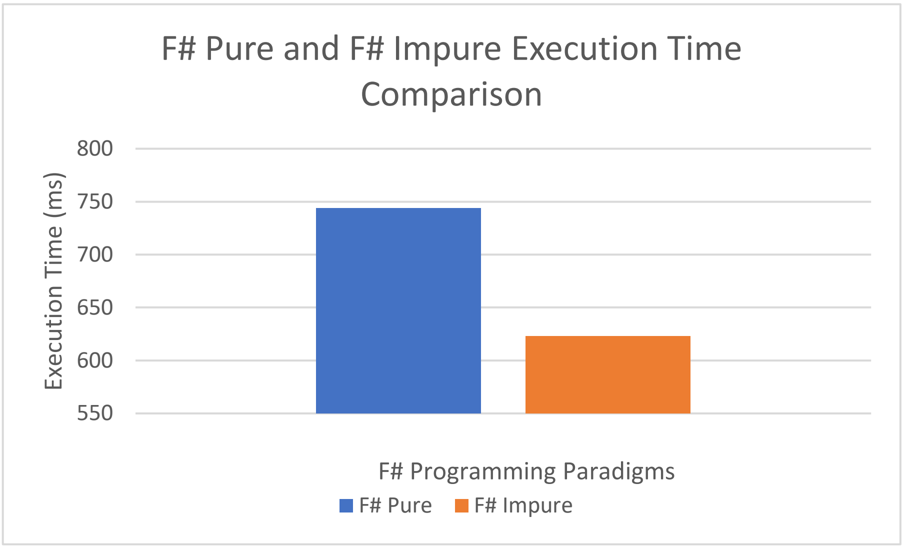


Figure 4: Comparison of Execution Time Between Pure and Impure MiniMax with Alpha-Beta Pruning

A difference in execution time is noticeable and the pure F# implementation had a greater execution time than the impure F# implementation, as hypothesized. This is due to the fact that the impure F# implementation allowed for mutability in state, reducing CPU and memory usage.

As the C# implementation was not completed, I am unable to present the differences in execution times. However, it can be hypothesized again, that the OOP implementation would be faster than the pure and impure F# implementation as OOP does not only allow for mutable states, but also iteration. The F# implementations consisted of a nested recursive function within the recursive *MiniMax* function. However, since C# allows for *breaks*, a recursive function would not be required. While recursive functions are elegant, iteration is often more efficient. Thus, C# would be more efficient in both execution time and memory.

Code Readability, and Concision

Code readability is defined by standards of indentation, formatting, and other coding syntax, and code concision. F# requires space indentation and white space around binary arithmetic expressions. As a result, there is much less coding “noise” such as braces, semicolons, and others. This style of coding allows for more concise code. F# also encourages reusability as functions are first class. There is also less manual type inference required due to the powerful type inference system of F#. In addition, F# syntax allows for much more self-contained and compact code by allowing syntax such as declaring record types to be composed in one line.

```
type Person = { FirstName: string; LastName: string; BornOn: DateTime }
```

Imperative F# code is also similarly readable, as most syntax remains the same, except the *mutable* keyword allowing mutable states.

C# with OOP can provide stronger readability due to the nature of the OOP paradigm. Moves and GameStates are much better to read and understand as objects according to the OOP implementation. However, C# can be more verbose than F# and can often seem more difficult to read.

Through the experience of programming in pure (F#), impure (F#), and OOP (C#) within this assignment, it was determined that F# provided better readability and concision.

For example, the TicTacToePure.fs contains concise record types for Player, Move, and GameState, as shown below,

```
type Player = Nought | Cross

type Move =
    { Row: int; Column: int }
    interface ITicTacToeMove with
        member this.Row with get() = this.Row
        member this.Col with get() = this.Column

type GameState =
    { Size: int; Turn: Player; Board: List<List<string>> }
    interface ITicTacToeGame<Player> with
        member this.Turn with get()    = this.Turn
        member this.Size with get()    = this.Size
        member this.getPiece(row, col) =
            this.Board.[row].[col]
```

Figure 5: Record Types – Player, Move, and GameState in TicTacToePure.fs

In the OOP implementation, Player, Move, and GameState were declared as objects. While this may be easily understandable, it is clear that the F# implementation is much more concise and compressed.

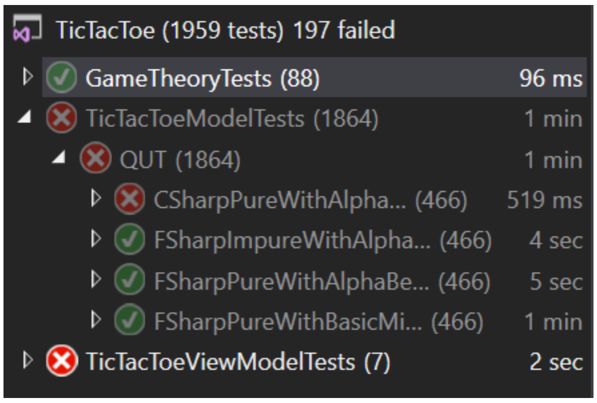
Finally, the lack of verbosity and type inference of F# also allows for greater readability, shortening the number of lines required, and also increasing efficiency.

Appendix

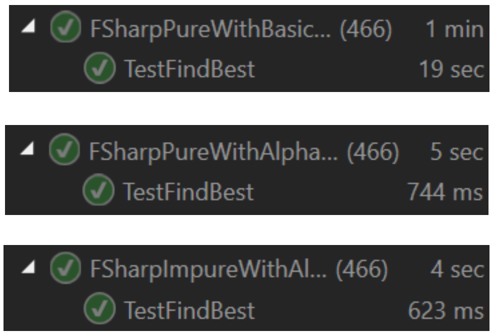
Test Analysis Specifications and Conditions

1. The algorithm analysis and testing were implemented with the F# and C# programming language through Visual Studio 2017.
2. The testing was performed on an Acer Spin SP513-51 laptop, running Microsoft Windows 10 Home, with an Intel Core i7-7500U CPU @ 2.70GHz. The tests were run in parallel using Visual Studio 2017.

Appendix 1: Test Environment Specifications

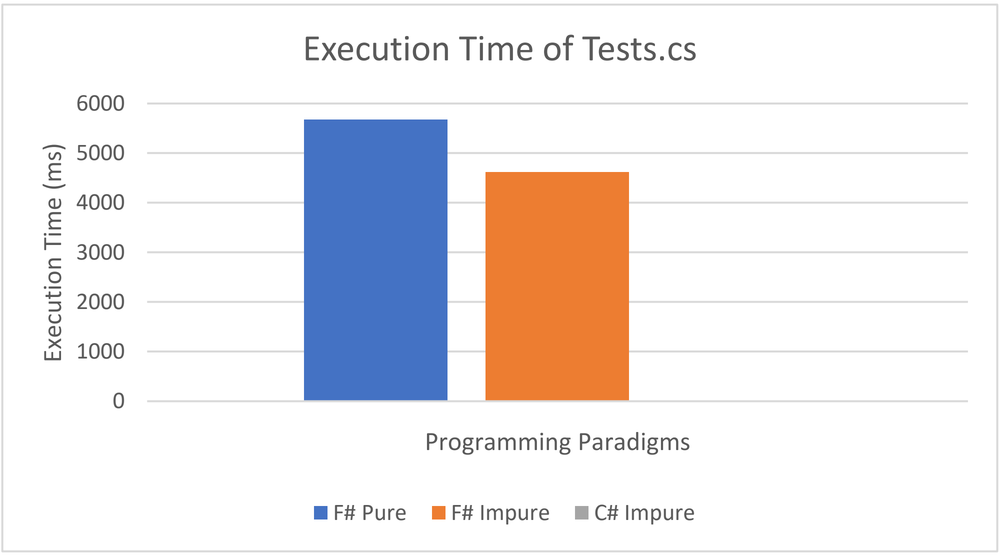


```
TicTacToe (1959 tests) 197 failed
  GameTheoryTests (88)                     96 ms
  TicTacToeModelTests (1864)               1 min
    QUT (1864)                             1 min
      CSharpPureWithAlpha... (466)        519 ms
      FSharpImpureWithAlpha... (466)       4 sec
      FSharpPureWithAlphaBe... (466)       5 sec
      FSharpPureWithBasicMi... (466)       1 min
  TicTacToeViewModelTests (7)              2 sec
```

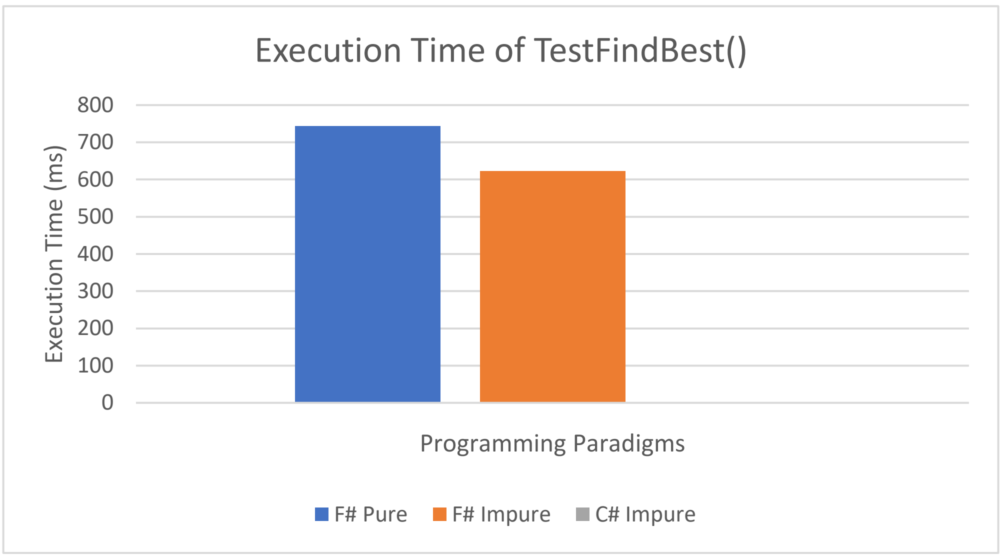


```
FSharpPureWithBasic... (466)    1 min
  TestFindBest                 19 sec

FSharpPureWithAlpha... (466)    5 sec
  TestFindBest                744 ms

FSharpImpureWithAl... (466)     4 sec
  TestFindBest                623 ms
```

Appendix 2: Test.cs Results and Corresponding Execution Times



Appendix 3: Execution Time of Test.cs Comparison Between F# Pure, F# Impure, and C# Impure. (C# Impure results are unavailable as the C# implementation was not completed)



Appendix 4: Execution Time of TestFindBest() Comparison Between F# Pure, F# Impure, and C# Impure