Higher order functions:-

One feature exclusive to functional programming is the concept of higher order functions. This type of function has one or more of the arguments as functions. This feature is implemented by currying. Functional programming does not make any difference between a value and a function. This feature can be used to create very complicated functions and reduce effort on the programmer’s part.

A common example is the map function:-

Let map f x = x f

The parameters themselves can be higher order functions.

Tail recursion:-

Tail recursion is a special form of recursion in which the recursive call occurs at the end of the function definition. A recursive function written in tail recursive form is always better than a function in which some expression occurs after the recursive call. The reason for this is that in case the recursive function is not in a tail recursive form, the compiler always reserves a set amount of memory as stack space for further calculations in each recursive step. Thus if there is very deep recursion (many recursive calls) it might be that free memory runs out and a exception arises thus the required result is not obtained. In the case of tail recursive functions, this never happens because the compiler knows that there is no computation after the recursive call is made and the value returned. Thus there is no stack space to be required and even with very deep recursion , the function runs normally without any problems.

An example of this is as follows:-

let rec count n =

if n = 1000000 then

printfn "done"

else

if n % 1000 = 0 then

printfn "n: %i" n

count (n + 1) (\* recursive call \*)

() //if these brackets did not exist, then this function would have been tail recursive

Logic evaluation:-

One of the most important applications of functional programming is logic evaluation (which is a vital part of proving theorems) Complicated logical expressions can be evaluated with ease using recursion and logic evaluation constructs.

1. In this program we shall develop a simple program that a compound logical statement.{Eg-Or(Not(And(True,Not True)),Or(Or(True,True),Not True))

}. The depth of the statement can be arbitrary. The program utilizes a recursive function which is as follows:-

type proposition =

| True

| Not of proposition

| And of proposition \* proposition

| Or of proposition \* proposition

let rec eval = function

| True -> true

| Not(prop) -> not (eval(prop))

| And(prop1, prop2) -> eval(prop1) && eval(prop2)

| Or(prop1, prop2) -> eval(prop1) || eval(prop2)

The logic of the program is as the follows:-

1. Define the logic expressions you need to evaluate in the code
2. Define the logic evaluator function as given above.
3. Pass the parameters ( A print statement along with the expression into the eval function)
4. The function recursively evaluates the answer. Print the answers in the end and exit

Limitations of program:-

1. Did not utilize any i/o functions to get input expression( the reason is that i/o is a integral part of imperative programming and we wished to implement a purely functional programming based logic evaluator)
2. Can run out of recursive stack space if not used with tail recursion.

Profiling results:-

Sample expressions used:-

1. Or(Not True,Not (Not True))
2. Or(Not(And(True,Not True)),Or(Or(True,True),Not True))

Results obtained:-

1. True
2. True

Asynchronous workflows and concurrency:-

Asynchronous workflows exist whenever the sequential synchronous run of instructions breaks. One particular version of this is concurrency or simultaneous parallel execution of multiple threads. Many problems arise if the programming is done in pure imperative of object oriented style. Central to this is the problem of mutability of data. Two very common problems arise in concurrency applications:-

Race Conditions:- Consider two threads T1 and T2. Both T1 and T2 run the following operations:-

Thread A Thread B

1A: Read variable V 1B: Read variable V

2A: Add 1 to variable V 2B: Add 1 to variable V

3A Write back to variable V 3B: Write back to variable V

There is no mechanism to actually predict which one of the threads is being executed earlier or later in real time. Thus If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition.

Deadlocks:-

In order to solve the problem of race conditions a programming construct called lock was created. Its basic idea is to ‘lock’ thread’s state so that another thread cannot use it erroneously.

Thread A Thread B

1A: Lock variable V 1B: Lock variable V

2A: Read variable V 2B: Read variable V

3A: Add 1 to variable V 3B: Add 1 to variable V

4A Write back to variable V 4B: Write back to variable V

5A: Lock variable V 5B: Lock variable V

This will ensure the previous problem will not occur as the variable V is blocked and it cannot be erroneously used by the two threads.

Now consider this situation. Let's say we have ThreadA and ThreadB which operate on two corresponding pieces of shared state, StateA and StateB. ThreadA locks stateA and stateB, ThreadB locks stateB and stateA. If the timing is right, when ThreadA needs to access stateB, its waits until ThreadB unlocks stateB; when ThreadB needs to access stateA, it can't proceed either since stateA is locked by ThreadA. Both threads mutually block one another, and they are unable to proceed any further. This is called a deadlock.

Central to both these errors are the fact that data used in instructions is mutable which means their values can be changed. When running two or more threads simultaneously erroneous updation of the data being referenced may occur. Functional programming emphasizes use of immutability of data, thus both the problems of race conditions and deadlocks can be removed. Thus parallel programming becomes much easier in functional programming. F# provides many programming constructs for the implementation of these features. These include:-

Async module:- The async module is a module which has many inbuilt methods and objects to support asynchronous workflows. We shall be using the following methods

1. Async.Parallel- Runs threads in parallel processor
2. Async.Run:-Initiates the running of the thread

Mailbox processors:- is essentially a dedicated message queue running on its own thread. Any thread can send the MailboxProcessor a message asynchronously or synchronously, allowing threads to communicate between one another through message passing. This is the way in which message passing interface is implemented in F#.

In this paper we shall develop a simple program that downloads five urls both asynchronously (parallel) and synchronously(using list.map module). We shall compute the times required for the entire procedure using the .net system.diagnostics module for both cases and compare the result.

The user defined module used for the program is defined in F# as follows:-

module Seq =

let pmap f l =

seq { for a in l -> async { return f a } }

|> Async.Parallel

|> Async.Run

|> is actually the pipeline operator which is defined as follows:-

let inline (|>) x f = f x

The logic of the program is as the follows:-

1. Declare the urls in the code.
2. Create a parallel map function and using the definition given above.
3. Define a function time that is a higher order function, has a function and a print message as parameters. It starts a System.Diagnostic stopwatch runs the parameter function and stops the stopwatch. It prints data about run of the function
4. Run the time function once with the pmap function described above and once with the List.map function, compare the results.

Profiling results:-

Urls used:-

1. http://www.facebook.com/
2. http://www.google.com/
3. http://craigslist.com/
4. http://www.globalstudentguide.com/
5. http://en.wikibooks.org/wiki/Main\_Page/

Results obtained:-

Start...

(4340.190980 ms) Synchronous: [("http://www.craigslist.com/", 185); ("http://www.google.com/", 262);

("http://en.wikibooks.org/wiki/Main\_Page", 190);

("http://www.globalstudentguide.com/", 132); ("http://facebook.com/", 296)]

(1939.11650 ms) Asynchronous: [|("http://www.craigslist.com/", 185); ("http://www.google.com/", 261);

("http://en.wikibooks.org/wiki/Main\_Page", 190);

("http://www.globalstudentguide.com/", 132); ("http://facebook.com/", 294)|]

Done.

The synchronous version gave almost a 2.3x boost over the asynchronous version. This computation was done in a 8 core Intel Xeon processor. 3 GHz each.

Limitations are similar to the previous problem.

References:-

1.http://en.wikibooks.org/wiki/F\_Sharp\_Programming

2.Beginning F# by Robert pickering, APRESS 2009

3.Introduction to functional programming,Richard Bird, Philip Wadler Prentice Hall, 1988