Originally published in the magazine dotNetMania Vol 4 Item 73. September 2010. http://www.dnmplus.net

Functional Representation of Numerical Matrixes

Horacio Nuñez[1]

Abstract

This article introduces a new data type programmed with F# to represent numerical matrixes that uses functions instead of numerical arrays. We describe how this new abstraction could contribute to reduce the consumption of memory while at the same time confer to the execution of operators a lazy evaluation and allow a programming style similar to mathematical notation. Furthermore we discuss how this features and the usage of the iterator pattern enables the expression of computations that serves as the input to Parallel LINQ. Lastly we present a customization of the tool F# Interactive that includes support for the new data type developed along the text.

1 Introduction

The representation of numerical matrixes by means of two-dimension arrays is a very extended practice taken the symbolic relation between the concept of lineal algebra and the semantic this structures offers. In the case of the named sparse matrixes there are also many conventions that take advantage of the great number of null values to reduce the cost of memory storage, thus improving the respond times. Although for a sparse matrix of dimensions great enough the memory footprint still could be excessive.

This article suggest a new abstraction that allow the work with matrixes without computing its elements entirely, just those elements needed at a given time, using for that a function that enable the resolution of an element based on its coordinates. For the implementation it was used the F# language whose functional and object oriented nature enable the encapsulation of routines and functional algorithms in a data type compatible with the CLI.

Once commented the data type we will analyze an immediate consequence of the usage of a function to identify a matrix. That is the chance of defining iterators that traverse a matrix's elements thus expressing in a declarative way computations that serves as the input to the PLINQ technology. Ultimately the reader will be presented with a customization of the tool F# Interactive to try the new data type.

2 Numerical Matrixes and Arrays

The usage of two-dimensional arrays is a widely used approach to program data type to represent numerical matrixes (in the following we only mention the term matrix). It's certainly hard to find another data structure that keeps a closer relation with the lineal algebra concept than this resource, which is so familiar for programmers and almost ubiquitous in any programming language. But this symbolic relation is not perfect in practice and the employ of arrays, though natural, adds to the resolution of numerical problems via matrixes important considerations about the coding style in the definition of an operation as well how much memory and time it requires to complete.

The finite character of the memory of any computer guarantee that the existences of an upper limit for a certain vector whose dimensions is great enough. Even if this affirmation could be seeing as extreme the intrinsic complexity of the world in which we live could easily provide problems that involves a finite but large number of variables that requires to be optimized by the usage of linear systems [1, 2].

A special type of matrix, named sparse have the particularity of include a high number of null elements, property that is taken into account in many actual conventions [3] to storage just those element different from the one who prevail. In this family of abstractions the usage of arrays has a more discrete presence and the upper limits are higher that those of two-dimensional arrays. Of course these benefits are only achieved if and only if we are truly in the presence of a sparse matrix.

Moreover operations between matrixes represented by arrays have to exhaust all the coordinates of the resulting matrix resolving the related elements before any other operation can take place. This process clearly increases the processing time and in those cases in which just a portion of the final matrix is relevant ends being partially needless. Besides this coding style more explicit differ from the mathematical notation which is more declarative.

[1] hnh12358@gmail.com, http://horatio.info

It will be possible to apply functional programming practices like lazy evaluation and declarative coding to the work with matrixes?

It could be possible to have an abstraction for a matrix that doesn't use arrays?

3 Functional Abstraction

The exercise of sparse matrixes conventions in the previous section is justified by the existence of an isomorphism between the vector space of this structures and the one of matrixes. This concept of linear algebra establish a relation one to one between two structures based on a bijective function (or application) that translates an element of one set to its equivalent in the other. Then it's valid to save resources using sparse representations for matrixes that match the criterion because the isomorphism will guarantee the bind between coordinates and its correspondent element.

Keeping that bind no matter what mechanism is used defines implicitly an isomorphic relation and we are in the presence of an abstraction to represent matrixes. Note that any abstraction of a matrix gets defined by the information of its dimensions and the function that retrieves its elements.

Now, if we think in the set of numerical functions of two variables belonging to the natural numbers we notice it's a vast and extremely versatile set. Consider the reader if taking the set of valid coordinates associated to a given matrix's dimensions it will be possible to find a function such that for each evaluation of coordinates it will yield the associated element? The answer is affirmative, and it's very important to highlight that such function doesn't have to be an exact reflect of the mathematical notation.

As an example take the set of the square matrixes whose elements are the natural numbers ordered in the normal way, fixing the order as 3 we have the following matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

The naïve construction it's based in the usage of conditionals attending the indexes of the row and columns as showed in **code 1 a**. Simple but effective, and not despicable if we take into account that not all numeric values are loaded in memory. However we can still construct a more efficient function for this matrix given that it shows a well know pattern. The functions of **code 1 b** and **code 1 c** are more smarter in this sense, in particular the function **simpleC** which considers the dimensions of the matrix (identifiers **rowCount** and **columnCount**) allowing the escalate of this kind of matrix to higher dimensions using the same function.

```
let simpleA (i,j) =
    if i = 1 then
       if j = 1 then 1
       elif j = 2 then 2
       elif j = 3 then 3
       else
        failwith "invalid column"
    elif i = 2 then
       if j = 1 then 4
       elif j = 2 then 5
       elif j = 3 then 6
       else
        failwith "invalid column"
    elif i = 3 then
       if j = 1 then 7
       elif j = 2 then 8
       elif j = 3 then 8
       else
        failwith "invalid column"
    else
        failwith "invalid row"
```

code 1 a

```
let simpleB (i,j) =
    if (i >= 1 && i <= 3) && (j >= 1 && j <= 3) then
        (i - 1) * 3 + j
    else
        failwith " invalid coordinates"
```

code 1 b

```
let simpleC (i,j) =
    if (i >= 1 && i <= rowCount) && (j >= 1 && j <= columnCount) then
        (i - 1) * rowCount + j
    else
        failwith "invalid coordinates"
```

code 1 c

Without question this approach is more economic than to store all the natural numbers in a two-dimensional array. Like this example we can found many other matrixes whose values could be resolved by the means of a straightforward function. Some well know examples are the matrixes: identity, null, Hankel, Hilbert and Vandermonde.

The trivial representation however is the proof of existence of this function that we shall name *generative* and justifies the implementation on the following sections of an abstraction purely functional where a matrix its defined by a tuple of the form:

(rows count, columns count, generative function)

4 The FMatrix data type

As was mentioned in the beginning of the text the F# language [4] stand out for introducing the functional paradigm over the CLI platform, that is, like in the case of other .NET languages with F# it's possible to define new data types that can be consumed from other languages of the platform like C# and Visual Basic. A point of convergence between the functional constructs and the inference system of F# is the feature called *implicit type construction* by which using a tuple we succinctly can define the fields that comprises the data type letting for the compiler the work of generating the appropriate constructors.

In **code 2** starts the implementation of the type **FMatrix** using this mechanism. Note that we don't have to specify the data type of the tuple's components **(rows, columns, gen)**. The F# compiler will infer then based on the usage in the rest of the code.

The first instruction of the body begins with the keyword **do** that allows adding for every constructor the invocation to an action that doesn't return a significant value. Here we execute the auxiliary function to set a precondition that prevents the construction of matrixs with negative dimensions. The motive about we do allow null dimensions will be explained shortly.

The next instruction declares a second constructor which third parameter is of the delegate type **Func<Int32,Int32,Double>**. This overload is needed since **gen** identifier, and consequently the third parameter of the first constructor are inferred of being of type **FSharpFunc** <**Tuple<Int32,Int32>,Double**>. A fact that will prevent other languages of creating new instances of this class by using their support for lambda expressions:

```
//C#
new FMatrix(128, 128, (i,j) => Math.Pow(-1, i + j));

'Visual Basic
New FMatrix(128, 128, Function(i,j) Math.Pow(-1, i + j))
```

This incompatibility is due because the way F# supports functions that can treated as data (commonly named as high order functions), it's doesn't employ delegates instead used subclasses of **FSharpFunc**<**T**, **TResult**> resolved by the compiler. The solution as can be observed is very simple and consists in calling the **Invoke** of the delegate from the body of a new lambda expression that will be passed as the third parameter of the first constructor.

```
type FMatrix(rows,columns,gen) =

    do requires (rows >= 0 && columns >= 0)  MATRIX_CREATION_BADDIMENSIONS

    new (rows, columns, gen :  Func<Int32,Int32,Double>) =
        requires (gen <> null) MATRIX_CREATION_NULLFUNCTION
        let newGen = (fun (i,j) -> gen.Invoke(i,j))
        new FMatrix(rows, columns, newGen)

    member public this.RowCount
           with get() = rows

    member public this.ColumnCount
           with get() = columns

    member public this.Dimension
           with get() = (rows,columns)

    member public this.Count
           with get() = rows * columns

    member public this.MainDiagonalCount
           with get() = Math.Min(rows,columns)

    member public this.IsSquare
           with get() = rows = columns
    member public this.IsEmpty
           with get() = rows = 0 || columns = 0

    member public this.IsRowVector
           with get() = rows = 1

    member public this.IsColumnVector
           with get() = columns = 1
```

code 2

The remained instructions declares a set of read only properties to bring information about the instance, the properties **RowCount** and b are necessary since the identifiers in an implicit type construction have private visibility. The identifiers of the class's members (keyword **member**) must be prefixed by an identifier and the dot character, the identifier will represent the current instance and it's not required to be the same for all members, however we chose **this** for affinity with C#.

5 Empty Matrixes

Before we start in the next section with the implementation of methods and operators its mandatory to introduce the idea of a matrix that doesn't contains any elements, say an empty matrix. Although is not a concept that can be visualized its definition: *a matrix in which at least one of the dimensions is equal to zero* have been proved to be consistent with the matrix theory introduced in the Algebra Lineal texts [5].

Likewise its theory relevance is that it become convenient to begin inductive formulations and proofs, for the perspective of a programmer it becomes a natural way to serve as the base case for recursive procedures. An example is the recursive calculation of a determinant via minors, where the case base is the determinant of a empty matrix of 0x0 which is accepted of being 1.

Next we reproduce the proposal of [6] to incorporate empty matrixes in the domain of basic operations like product by a scalar (1), sum of matrixes (2), product of matrixes (3-5) as well as other relations (6-7).

$$1)\quad \alpha \cdot [\,]_{p\times m} = [\,]_{p\times m} \cdot \alpha = [\,]_{p\times m}$$

$$2)\quad [\,]_{p\times m} + [\,]_{p\times m} = [\,]_{p\times m}$$

$$3)\quad [\,]_{0\times m} \cdot \mathrm{X}_{m\times p} = [\,]_{0\times p}$$

$$4)\quad \mathrm{X}_{m\times p} \cdot \mathrm{X}_{p\times 0} = [\,]_{m\times 0}$$

$$5)\quad [\,]_{p\times 0} \cdot [\,]_{0\times m} = 0_{p\times m}$$

$$6)\quad [\,]_{p\times m} = 0_{p\times m}$$

$$7)\quad [\,]_{0\times 0} = \mathrm{I}_{0\times 0} = (\,[\,]_{0\times 0})^{-1}$$

Please note that the objective of this formalization is not to roll the dice on certain propositions but allow the works with empty matrixes respecting the usual requirements of any operation. For example, it doesn't make sense sum a 5x5 matrix with a 0x0 matrix because they don't have equal dimensions. Missing this and establishing that such operation will yield the 5x5 operation we are introducing a violation in the contract of the sum operation, which will lead us in an over checking in future procedures.

6 Elementary Operations

Now we will discuss the implementation of elemental operation in the data type **FMatrix** while commenting its implications in the functional work with matrixes. The idea behind every procedure is to create a new generative function that stands over the previous ones.

1. *Resolution and changed of an element based on its coordinates:*
   To obtain an element based on its coordinates F# supports the declaration of indexers through the declaration of an **Item** property, as can be seen in **code 3**. Please note in which the elements of the matrix will be resolved just when they are required and what this happens they are not saved anywhere thus preventing the over consume of memory. Since we are talking about an immutable type to substitute an element it's necessary to create a new instance. Thus it's impossible to define a **set** block within the **Item** property because it will not return **unit as**

required but and an instance of **FMatrix**. To overcome this it's necessary to define a new function named **ChangeItem** that will create a new generative function with a proper condition that in case of evaluate to false will rely on the current indexer function, that's is the original generative function.

```
member public this.Item
       with get (i,j) =
            requires (this.AreValidIndexes(i,j)) MATRIX_INDEXER_BADINDEXES
            gen(i,j)

member public this.AreValidIndexes (i,j) =
       (i >= 1 && i <= rows && j >= 1 && j <= columns)

member public this.ChangeItem(w,z,value) =
       requires (this.AreValidIndexes(w,z)) MATRIX_INDEXER_BADINDEXES
       let newGen = fun (i,j) -> if (i,j) = (w,z) then value else this.[i,j]
       new FMatrix(rows, columns, newGen)
```

code 3

2. *Rows and columns permutation:*
   In **code 4** the new generative function again uses a conditional to invert the coordinates. Note the reader that the parameter **r1** (**c1**) y **r2** (**c2**) doesn't not belong to the execution context of the new function but nevertheless they are used in its definition. In the presence of this situation it's said that the parameters are captured by the closure of the lambda expression.

```
member public this.PermuteRows(r1,r2) =
              requires (r1 >= 0 && r1 <= rows && r2 >= 0 && r2 <= rows)
                       MATRIX_PERMUTEROWS_BADINDEXES
              let newGen = fun (i,j) -> if i = r1 then this.[r2,j]
                                        elif i = r2 then this.[r1,j]
                                        else this.[i,j]
              new FMatrix(rows,columns,newGen)

member public this.PermuteColumns(c1, c2) =
              requires (c1 >= 0 && c1 <= columns && c2 >= 0 && c2 <= columns)
                       MATRIX_PERMUTECOLUMNS_BADINDEXES
              let newGen = fun (i,j) -> if j = c1 then this.[i,c2]
                                        elif j = c2 then this.[i,c1]
                                        else this.[i,j]
              new FMatrix(rows,columns, newGen)
```

code 4

3. *Transposed Matrix:*
   The generative function of the transposed matrix in **code 5** stands out for its length and above all for its similarity with the usual mathematical notation.

```
member public this.Transpose() =
       let newGen = fun (i,j) -> this.[j,i]
       new FMatrix(columns,rows,newGen)
```

code 5

4. *Horizontal and Vertical concatenation:*
   In **code 6** can be seen how both concatenations impose conditions over the dimensions in the involved matrixes. Once validated the operation the resulting matrix will have a bigger dimension and the new generative function will rely on the two previous. The usage of the methods **ConcatHorizontal** and **ConcatHorizontal** can be very useful in such cases in which we have to create matrix made of blocks whose generative function have know expressions but none the result of the concatenation.

```
    member public this.ConcatHorizontal(a : FMatrix) =
           requires (rows = a.RowCount) MATRIX_CONCATHORIZONTAL_MISMATCH
           let newGen = if this.IsEmpty && a.IsEmpty then
                           fun (i,j) -> failwith MATRIX_EMPTY
                        else
                           fun (i,j) -> if j > columns then a.[i,j - columns]
                                        else this.[i,j]
           new FMatrix(rows, columns + a.ColumnCount, newGen)

    member public this.ConcatVertical(a : FMatrix) =
           requires (columns = a.ColumnCount) MATRIX_CONCATVERTICAL_MISMATCH
           let newGen = if this.IsEmpty && a.IsEmpty then
                           fun (i,j) -> failwith MATRIX_EMPTY
                        else
                           fun (i,j) -> if i > rows then a.[i - rows,j]
                                        else this.[i,j]
           new FMatrix(rows + a.RowCount,columns,newGen)
```

code 6

5. *Sum of matrixes:*
   Likewise the generative of the transposed matrix the function that comprises the sum operation is similar to the notation found in the algebra literature. As the first instruction of **code 7** uses a property of type **FMatrix** it's necessary to state explicitly the data type because the inference system doesn't have enough information to restrict the parameters to a single data type. Although the syntax for operator overloading in F# is similar to the ones of C# and Visual Basic, the F# compiler allows the definition of symbolic operator much more arbitrary they will always be usable from the very F#.

```
    static member public (+) (a : FMatrix,b : FMatrix) =
           requires (a.Dimension = b.Dimension) MATRIX_SUM_MISMATCH
           if a.IsEmpty then
              a
           else
              let newGen = fun (i,j) -> a.[i,j] + b.[i,j]
              new FMatrix(a.RowCount,a.ColumnCount, newGen)
```

code 7

6. *Product of a Matrix for an Scalar:*
   Similar to the previous operation the implementation it's more faithfully to the math notation. This time in the first method of **code 8** the compiler infers the **scalar** parameter to be of type **System.Double** because its usage in the new generative function. On the second method we do have to provide the data type to relying on the first one.

```
    static member public (*) (scalar,a : FMatrix) =
           if a.IsEmpty then
              a
           else
              let newGen = fun (i,j) -> a.[i,j] * scalar
              new FMatrix(a.RowCount,a.ColumnCount, newGen)

    static member public (*) (a : FMatrix,scalar : Double) =
           scalar * a
```

code 8

7. *Matrix Multiplication:*
   Without doubts for this operation the new abstraction has as principal appeal the fact that only the required element will be calculated, reducing or equaling (depending of the program) the cubic order we get using arrays. Also we get a concise code since in **code 9** once resolved the cases involving empty matrixes the new generative function it's a translation of the Sum symbol that defines the element in the coordinate (i, j):

$$\mathbf{X} * \mathbf{Y} = \sum_{\mathbf{n=1}}^{\mathbf{Y.RowCount}} \mathbf{Y}.[\mathbf{i},\mathbf{n}] * \mathbf{Y}.[\mathbf{j},\mathbf{n}]$$

The instruction **seq {1 .. b.RowCount}** is a range expression that serves as shorthand to produce a succession of natural numbers enumerating the row indices of the right matrix. The consolidation operator **fold** processes this numbers and calculates the total using an anonymous function that matches the previous formula. Since the elements of the matrix are of type **System.Double** the second parameter of the **fold** must be a literal **0.0**.

An alternative of the traditional formula is the Strassen algorithm [7], the interested reader can found its implementation in the source code that join this article.

```
    static member public (*) (a : FMatrix,b : FMatrix) =
           requires (a.ColumnCount = b.RowCount) MATRIX_MULTIPLICATION_MISMATCH
           let newGen = if (a.IsEmpty && b.IsEmpty) then
                           fun (i,j) -> 0.0
                        elif (a.IsEmpty || b.IsEmpty) then
                           fun (i,j) -> failwith MATRIX_EMPTY
                        else
                           fun (i,j) -> seq {1 .. b.RowCount}
                                         |> Seq.fold (fun accu c -> accu + a.[i,c] * b.[j,c]) 0.0
           new FMatrix(a.RowCount,b.ColumnCount,newGen)
```

code 9

8. *Sub matrix Extraction:*
Even when the definition of a sub matrix comes naturally from the same matrix concept itself there are few programming languages that offer an easy syntax to express this idea. Fortunately this is not the case of F# and as happens with the indexers we saw before the compiler support convention to translate a simple syntax into the call to a specific method. In the **table 1** can be saw some example of this convention defining a rectangular portion using four optional values.

Expressions	*Generated Code*
matrix.[**starting row** .. **final row**,*]	matrix.GetSlice(Some(**starting row**),Some(**final row**),None,None)
matrix.[**starting row**.. , .. **final column**]	matrix.GetSlice(Some(**starting row**),None,**None**,Some(**final column**)
matrix.[... **final row, starting column** ..]	matrix.GetSlice(None,Some(**starting row**),Some(**starting column**),None)
matrix.[***,***]	matrix.GetSlice(None,None,None,None)

Table 1

The convention demand a property named GetSlice such that its parameters are values of the discriminated union **Option**, a data type that allow the expression of optional values that in this case will be of **System.Int32**. The instruction: **matrix.[3 .. , 10 .. 30]** will be resolved by the compiler as follows:

matrix.GetSlice(Some(3), None, Some(10), Some(30))

Where the lack of a third value indicates that must be extracted a sub matrix that includes all the elements in the intersection of all the rows but the first two and the column from the tenth and thirtieth. In **code 10** of can be found an inner function called **adaptSlice** that using a **match … with** resolve in four lines of code al the possible cases. To process with ease the dichotomy of the **Option** type F# supports a much appreciated feature called pattern matching, this construct has such a great potential and deserve an article on his own to be explained, however the reader is encourage to compare a similar code written in C# or Visual Basic.

```
member public this.GetSlice
       with get (rowStart,rowEnd,columnStart,columnEnd) =
            let areValidSlices (r1,r2,c1,c2) =
                r1 >= 0 && c1 >= 0 && r2 - r1 >= 0 && c2 - c1 >= 0 && r2 <= rows && c2 <= columns

            let adaptSlice first second count =
                match first, second with
                | Some(p1), Some(p2) -> (p1, p2)
                | Some(p1), None -> (p1, count)
                | None, Some(p2) -> (1, p2)
                | None, None -> (1, count)

            let (r1,r2), (c1,c2) = (adaptSlice rowStart rowEnd rows),
                                   (adaptSlice  columnStart columnEnd columns)
            requires (areValidSlices (r1,r2,c1,c2)) MATRIX_SUBMATRIX_INVALIDBOUNDS
            let newGen = fun (i,j) -> this.[r1 + i - 1, c1 + j - 1]
            new FMatrix(r2 - r1 + 1,c2 - c1 + 1,newGen)
```

code 10

9. *Construction of the generative function:*
   As was mentioned in the beginning given any matrix is possible to construct a generative function using an scheme based on conditionals. Logically this proof is useless if we don't provide an implementation to produce such a scheme, and to keep the line of the article so far it will be desirable that this code will be functional. To introduce such algorithm lets analyze the code in **code 11** in which is defined a generative function for a square matrix with the four first natural numbers.

   The associated function to the identifier **simpleA** could be found similar to the one of **code 1** but for the **elif** sentences have been replaced to anonymous function whose are immediately evaluated with the parameter of the function that invoke it. To make the code more readable in each declaration of an anonymous function, although is not necessary, was added to the identifier the numeric index in the matrix.

```
let simpleA (i,j) =
    (fun (i1,j) ->
        if i1 = 1 then
           (fun j1 -> if j1 = 1 then
                         1.0
                      else
                         (fun j2 -> if j2 = 2 then
                                       2.0
                                    else
                                       (fun _ -> failwith "end-of-row") j2 ) j1 ) j
        else
           (fun (i2,j) ->
                if i2 = 2 then
                   (fun j1 -> if j1 = 1 then
                                 3.0
                              else
                                 (fun j2 -> if j2 = 2 then
                                               4.0
                                            else
                                               (fun _ -> failwith "end-of-row") j2 ) j1 ) j
                else
                   (fun (_,_) -> failwith "end-of-matrix") (i2,j) ) (i1,j)) (i,j)
```

code 11

   Note that each anonymous function except those who emit an error message invoke within its body the function that represents the next coordinate, not matter if it's a row or a column. If we extract every anonymous function as a parameter that has to be submitted, the corresponding anonymous of the first row and the first value of the matrix could be written as follows:

```
let row1 value11Gen nextRow  =
    fun (i,j) -> if i = 1 then value11Gen(j) else nextRow(i,j)

let value11 nextColumn =
    fun j -> if j = 1 then 1.0 else nextColumn(j)
```

Repeating this process and notation in each of the anonymous function it’s possible to rewrite the function **simpleA** as follows:

```
let simpleA =
    (row1 (value11(value12(endOfColumn))) (row2 (value21(value22(endOfColumn))) endOfRows))
```

The importat observation with this example is that the generative function we are looking for can be constructed by the means of small function which only parameter if another function and its result type is in return a function that accepts one or two coordinates, note here that the signature of the function that must be passed as parameter is the same as the resulting function of that parameter passing. Clearly we have to delimit two families of this kind of functions, those who pass over the rows and those who pass over its values, the columns.

In **code 12** can be see this idea in practice. The function **createGen** has as first parameter an optional value that indicates if the conditional scheme must be optimized to reflect the repeat of a value in the matrix[2], the second parameter is expected to be a sequence of sequences of real numbers that characterize the matrix expanding it by rows. In the first instructions the functions that will serve as starting and finally value for the generation of the function that will represent rows and columns. Note that in the presence of a sparse matrix instead of the function that throw error message the final function will yield the non sparse value.

The idea that was extracted by looking into the function **simpleA** must be clearer after seeing the implementation of the functions **buildMatrixGen** and **buildRowGen**, which use the operator **foldic**[3] and the functions **matrixFold** y **rowFold** respectively to build the final generative function. In both cases the final result is the consolidation of a function that takes a single function parameter, so for each iteration must be defined another anonymous function that accept a function as the first parameter. This way once finished the consolidation it’s only needed to apply the results of the functions identified by **lastMatrixGen** and **lastRowGen**, which will finally yield functions that accepts coordinates. Note the reader that besides composing little functions the use of the operator **foldic** allows the verification that rows with different element count ends forming an **FMatrix** instance.

```
[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module FMatrix =

    let private createGen sparseValue rows =

        let isSparsed = Option.isSome(sparseValue)
        let sparsev = if isSparsed then
                         Option.get(sparseValue)
                      else
                         Double.NaN

        let firstRowGen nextGen = (fun j -> nextGen(j))
        let lastRowGen = if isSparsed then
                            fun _ -> sparsev
                         else
                            fun _ -> failwith "end-of-row"

        let firstMatrixGen nextGen = (fun (i,j) -> nextGen(i,j))
        let lastMatrixGen = (fun (_,_) -> failwith "end-of-matrix")

        let rowFold index previousGen value =
            if isSparsed && value = sparsev then
                previousGen
            else
                let currentGen elseGen =
                    fun j -> if j = index + 1 then
                                value
                             else
                               elseGen(j)
                (fun nextGen -> nextGen |> currentGen |> previousGen)
```

[2] Although we understand for sparse matrixes those who had many null elements the same considerations can be taken in the presence of another element that is much repeated.

[3] This function isn’t part of the F# distribution but it’s defined in terms of the operator **fold** extending it to associate every element with its index in the sequence and at the end of the execution also yield the number of processed items.

```
        let buildRowGen sequence =
            let count, currentRowGen = PSeq.foldic rowFold firstRowGen sequence
            let finalGen = lastRowGen |> currentRowGen
            count, finalGen

        let matrixFold index (previousColumnCountOption,previousGen) row =
            let currentColumnCount, currentRowGen = buildRowGen row
            let currentGen elseGen =
                fun (i,j) -> if i = index + 1 then
                                currentRowGen(j)
                             else
                                elseGen(i,j)

            match previousColumnCountOption with
            | None -> Some(currentColumnCount), (fun nextGen -> nextGen |> currentGen |> previousGen)

            | Some (previousColumnCount) when previousColumnCount = currentColumnCount ->
                    Some(currentColumnCount), (fun nextRowGen -> nextRowGen |> currentGen |> previousGen)

            | _ -> failwith MATRIX_GENCREATION_COLUMNCOUNTMISMATCH


        let buildMatrixGen rows =
            let rowCount, (columnCountOption,currentMatrixGen) = PSeq.foldic matrixFold (None,firstMatrixGen) rows
            let finalMatrixGen = lastMatrixGen |> currentMatrixGen
            rowCount,Option.get(columnCountOption),finalMatrixGen

        let rowCount,columnCount,matrixGen = buildMatrixGen rows
        new FMatrix(rowCount,columnCount,matrixGen)


    let fromSeq source =
        createGen None source

    let fromSeqSparse (sparseValue,source) =
        createGen (Some(sparseValue)) source
```

code 12

The function **createGen** differs from the rest of the function we have outlined in that it's not part of the **FMatrix** class, instead it's defined in the module of the same name. A module in F# slightly resemble the modules of Visual Basic since both allows the group of functionality that is no associated with any instance of a data type and in compile time are resolved as static classes with names match the names of the module. Because of the later the **createGen** function is tagged with the attribute **CompilationRepresentation** specifying that once compiled the suffix *Module* will be added to avoid a collision with the **FMatrix** data type. From the point of view of other .NET languages there indeed will be difference between the two compiled classes while for F# it gets unnoticed. The remaining functions **fromSeq** and **fromSeqSparse** along with other public functions of the module that depends of **createGen** supports the creation of functional matrixes instances from strings and an expansion of rows.

7 The iterator pattern and Parallel LINQ

So far we have guarantee an immutable data type in which all the operations take a constant time to return, but not the obtaining of these results. To continue the application of functional practices in the type **FMatrix** we will use at this point the iterator pattern. The classes that exposed this pattern can be recognized by implementing the interface **IEnumerable<T>**, which in F# are named *sequences*.

The mechanism of iteration this pattern offers is also lazy and assure that there is only one given element in the memory while iterating the sequence. In the particular of the **FMatrix** type this turns out to be a very useful tool to guarantee that no matter the length and complexity of a route within matrix's elements its iteration will not ask more memory that the requested for the more demanding element. In **code 13** appears the implementation of properties like **Items**, **ItemsIndexed** and **Rows** that uses *sequence expressions* to build iterators and have as results objects of type **IEnumerable<Double>**, **IEnumerable<Int32,Int32,Double>**, **IEnumerable<IEnumerable<Double>>** respectively.

On the other hand the method **GetItemsBy** receive as input a sequence of coordinates and produce a sequence with the corresponding elements.

```
    member public this.Items
            with get() =
                 seq { for i = 1 to rows do
                           for j = 1 to columns do
                               yield this.[i,j] }

     member public this.ItemsIndexed
            with get() =
                 seq { for i = 1 to rows do
                           for j = 1 to columns do
                               yield (i,j,this.[i,j]) }
     member public this.Rows
            with get() =
                 seq { for i = 1 to rows do
                         yield seq { for j = 1 to columns do
                                         yield this.[i,j] }}

     member public this.GetItemsBy(c) =
                   c |> Seq.map(fun (i,j) -> this.[i,j])
```

code 13

An instant benefit of using this Pattern is the possibility to use the **PLINQ** technology to express computation that involves the elements of a matrix. Being an immutable type it become less complex to use the own query operators of **Parallel LINQ** [8]. Note that since we are dealing with numerical matrixes of real numbers operations like product an algebraic sum comply with the associative property, thus making it easy to rely on aggregate operators like **Aggregate** and **Sum** in its parallel version. For example to compute the trace of matrix from C# using **PLINQ** we could use:

```
var coordinates = from x in Enumerable.Range(1, Math.Min(m.RowCount, m.ColumnCount))
                  select new Tuple<int, int>(x, x);
m.GetItemsBy(coordinates).AsParallel().Sum();
```

Which once it's executed will keep very constant memory consumption. Not following the C# and Visual Basic the F# grammar doesn't include keywords to match the standard query operators of **LINQ**, instead F# programmers can use the **Seq** module and its parallel version **PSeq**.

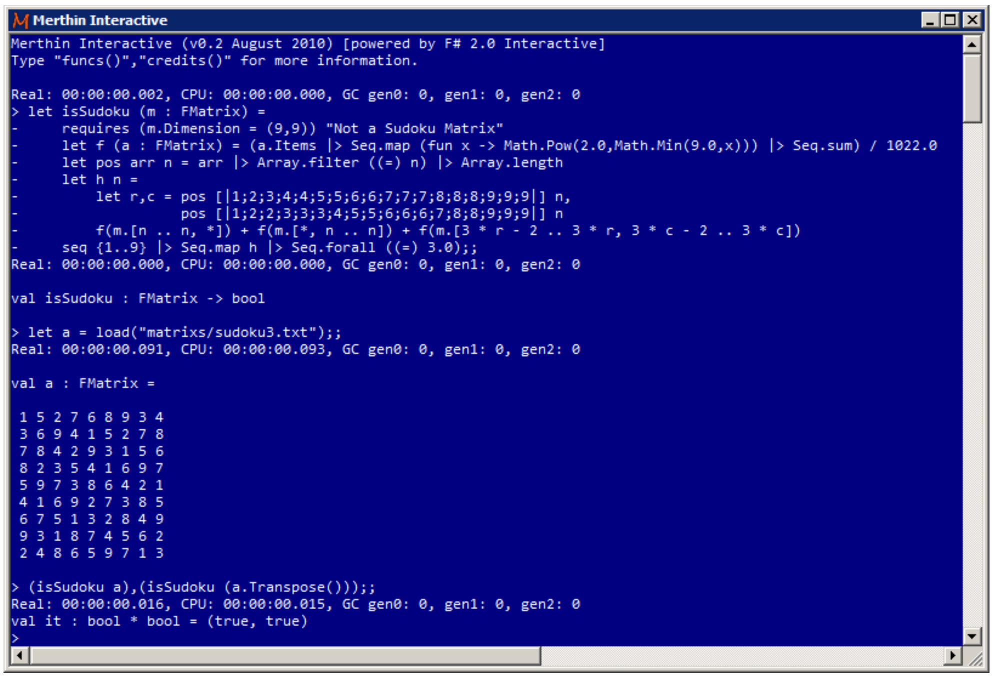
In **code 13** can be found the implementation of the **Equals** method, which determines whether an object in the current instance represents identical matrixs. In this code we rely on the **PSeq** module to create a sequence of Booleans values asserting the equality of items in the same coordinates. The **forall** operator ensures that once a false value is yielded the function will exit the possible lengthy operation with the negative result. It's worth mention that in case of run this code in a computer with a single core it will behave like if the **Seq** module was used instead.

```
override this.Equals (a) =
         match a with
         | :? FMatrix as fa when fa.Dimension = this.Dimension ->
              if fa.IsEmpty then
                 true
              else
                 seq { for i = 1 to rows do
                           for j = 1 to columns do
                               yield fa.[i,j] = this.[i,j] }
                 |> PSeq.forall((=) true)
         | _ -> false
```

listado 13

8 Merthin Interactive

The distribution of the F# language includes besides the compiler an REPL console called F# Interactive that after receiving F# code proceed to compiled it, executing it and output the results back to the console. This kind of programs turned to be very useful on evaluating small snippets of code like performing calculations and experimenting with the language. Additionally this tool allows the suggestion of an F# script (extension **.fsx**) to be executed before the user can type in the console. In the joint code of this article we used this option to prepare the interactive environment for calculations using the new data type. **Figure 1** shows the resulting console executing and showing the custom display of the **FMatrix** data type.



```
Merthin Interactive (v0.2 August 2010) [powered by F# 2.0 Interactive]
Type "funcs()","credits()" for more information.

Real: 00:00:00.002, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
> let isSudoku (m : FMatrix) =
-     requires (m.Dimension = (9,9)) "Not a Sudoku Matrix"
-     let f (a : FMatrix) = (a.Items |> Seq.map (fun x -> Math.Pow(2.0,Math.Min(9.0,x))) |> Seq.sum) / 1022.0
-     let pos arr n = arr |> Array.filter ((=) n) |> Array.length
-     let h n =
-         let r,c = pos [|1;2;3;4;4;5;5;6;6;7;7;7;8;8;8;9;9;9|] n,
-                   pos [|1;2;2;3;3;3;4;5;5;6;6;6;7;8;8;9;9;9|] n
-         f(m.[n .. n, *]) + f(m.[*, n .. n]) + f(m.[3 * r - 2 .. 3 * r, 3 * c - 2 .. 3 * c])
-     seq {1..9} |> Seq.map h |> Seq.forall ((=) 3.0);;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0

val isSudoku : FMatrix -> bool

> let a = load("matrixs/sudoku3.txt");;
Real: 00:00:00.091, CPU: 00:00:00.093, GC gen0: 0, gen1: 0, gen2: 0

val a : FMatrix =

 1 5 2 7 6 8 9 3 4
 3 6 9 4 1 5 2 7 8
 7 8 4 2 9 3 1 5 6
 8 2 3 5 4 1 6 9 7
 5 9 7 3 8 6 4 2 1
 4 1 6 9 2 7 3 8 5
 6 7 5 1 3 2 8 4 9
 9 3 1 8 7 4 5 6 2
 2 4 8 6 5 9 7 1 3

> (isSudoku a),(isSudoku (a.Transpose()));;
Real: 00:00:00.016, CPU: 00:00:00.015, GC gen0: 0, gen1: 0, gen2: 0
val it : bool * bool = (true, true)
>
```

Figure 1. Merthin Interactive

9 Conclusions

In this article was introduced a new data type for the work with numerical matrixes, that applying a purely functional approach attain the reduction in some degree of the finite memory problem and response times adjusted to the results that are really required. Likewise the reconciliation between code and math notation proved to be a very desirable feature that is obtained in a natural way using the new abstraction. This new approach is not intended as a proposition to substitute the current ones but a compliment to resolve matrix problems from another point of view.

A mandatory revision of this work that came become the motive behind future articles on F# is the usage of the expression trees of .NET 4.0 to represent both the generative function and elements of the matrix. The former has obvious implications on generative function optimization while the latter is essential to program operation of linear algebra where matrixes are made of polynomials (also known as λ-matrixes).

Another pending subject that surely the reader has notice is the no usage of generics to tailor the generative functions to adapt so that they might return other numeric data types. The reason of this absence is due because is not possible to constrain a type parameter so that it have to support operators. The correct solution will be to implement in F# the notions of Fields and Rings made by generic types but that will demand a larger text.

The motivation behind the project illustrated in this article was to experiment with the first official release of F# under the instruments of Visual Studio 2010. I hope that after reading this text the reader feels moved to try the new alternative this language offers and thus check how convenient it become to resolve more than complex program in an simple and elegant way.

The solution that contains the codes presented in the article can be download from the project repository at GitHub (www.github.com/hnh12358/Merthin).

References

[1] Kurosh, A. G. *Curso de Álgebra Superior 4^{ed}*. Editorial MIR, 1977

[2] Noriega, Teresita, de Arazoza, Héctor. *Álgebra Tomo 1 2^{ed}*. Editorial Félix Varela, 2003

[3] “Sparse Matrix”. http://en.wikipedia.org/wiki/Sparse_matrix

[4] Smith, Chris. *Programming F#*. O’Reilly, 2009

[5] De Boor, Carl. *An Empty Exercise*, SIGNUM, Volume 25, 1990

[6] C. N. Nett, W. M. Haddad. *A System-Theoretic Appropriate Realization of the Empty Matrix Concept*, IEEE Transactions on Automatic Control, Volume 38, Number 5, May 1993

[7] Cormen, Thomas H. et all. *Introduction to Algorithms 3*^{ed}. The MIT Press, 2009

[8] Duffy, Joe. Essey, Ed. *Running Queries On Multi-Core Processors*. MSDN Magazine, October 2007