Présentation F#

September 25, 2019

Table of Contents

[1]:
```
#I "./.fake/fsharp_ANF.fsx"
#load "./.fake/fsharp_ANF.fsx/intellisense.fsx"
#load "paket-files/mndrake/IfSharpLab/src/DeedleFormat.fs"
#load "XPlot.Plotly.fsx"
#I "../RProvider/bin"
#r "DynamicInterop.dll"
#r "RDotNet.dll"
#r "RDotNet.FSharp.dll"
#r "RProvider.dll"
#r "RProvider.Runtime.dll"
#fsioutput "on"
```

1 Un peu d’histoire

(Syme, 2019)

Année

Recherche

“Corporate”

1973

ML, le Lisp « typé »

1975

Microsoft

1995

Java

1997

Standard ML

1998

Don Syme @ Microsoft Research

2002

.NET (Kennedy and Syme, 2001)

2005

F#

SML : spécification formelle JAVA : vague de l'OO mais technos venant de la FP : (GC, JIT, bytecode)

2 F# et .NET

- F# est vu comme LE langage fonctionnel par MS sur .NET
- .NET et F# ont co-évolué

3 F# et .NET core

- Reboot .NET
- 100% multiplateforme (Mac/Windows/Linux + x86/ARM)
- OSS

4 F# Vs < X >

4.1 Avantages

- Stable
- Cohérent
- Ouvert
- Interopérable

4.2 Inconvénients

- Fonctionnalités
- Pas de "killer app"
- Couverture académique

- stable : pas de reboot/changement dans le langage, bonne "compatibilité" binaire
- Il y a une façon de programmer en f# relativement simple, pas de prise de tête sur 30 manières de coder la même chose
- gros des troupes chez MS, mais participation active et ouverte par RFC à l'évolution du langage

5 Utilisation dans l'industrie

- MS, of course (Cloud et produits internes)
- Jet.com
- Genetec
- Banques et Trading (ingénierie de la finance)
- Développement mobile (Xamarin)

- site de vente en ligne concurrent d'amazon, succès (vente - cher et plus local)
- Genetec : grosse boîte de soft, spécialisé dans la vidéosurveillance (leader)

6 Du code du code du code !

6.1 Les Fournisseurs de Types (*Type Providers*)

(Syme, Battocchi et al., 2012)

Un fournisseur de type F# est un composant qui fournit des types, des propriétés et des méthodes. Les Fournisseurs de type génèrent ce que l'on appelle des *Types fournis*, qui sont générés par le compilateur de `F#` et sont basés sur une source de données externe.

Par exemple, un fournisseur de Type pour SQL peut générer des types représentant les tables et colonnes dans une base de données relationnelle. En fait, c'est ce que fait `SQLProvider`.

Les types fournis dépendent dépendent des paramètres d'entrée d'un fournisseur de Type. Ces paramètres peuvent être un échantillon de source de données (par exemple, un fichier JSON), une URL qui pointe directement vers un service externe ou une chaîne de connexion à une source de données.

Un fournisseur de Type peut également vous garantir que les groupes de types sont "instanciés" uniquement à la demande ; Autrement dit, ils ne sont appelés que si les types sont réellement référencés par votre programme. Cela permet l'intégration directe et à la demande d'espaces de données à grande échelle, tels que les marchés de données (*data market*) en ligne, de manière fortement typée.

6.1.1 Fournisseurs de type "génératif"

Les fournisseurs de Type générative produisent des types qui peuvent être écrits en tant que types .NET dans l'assembly dans lequel ils sont générés. Cela leur permet d'être utilisé à partir du code dans d'autres assemblys. Cela signifie que la représentation typée de la source de données doit être généralement facile à représenter avec des types .NET.

6.1.2 Fournisseurs de type "effacé"

L'effacement (*erasure*) des fournisseurs de Type produisent des types qui peuvent uniquement être utilisées dans l'assembly ou le projet à partir duquel ils sont générés. Les types sont éphémères ; Autrement dit, ils ne sont pas écrites dans un assembly et ne peut pas être consommés par le code dans d'autres assemblys. Elles peuvent contenir des membres retardés (*delayed*), ce qui permet de passer à l'échelle facilement. Elles sont utiles pour l'utilisation d'un petit sous-ensemble d'une source de données volumineux et interconnectées.

6.1.3 Les Fournisseurs de Type les plus courants

`FSharp.Data` inclut les fournisseurs de Type pour les formats de document JSON, XML, CSV, HTML et ressources.

`SQLProvider` fournit l'accès fortement typé aux SGBD via le mappage de l'objet et F# requêtes LINQ sur ces sources de données.

6.2 Type Providers I : ce bon vieux CSV...

[2]:
```
open FSharp.Data

type Gbpusd = CsvProvider<const(__SOURCE_DIRECTORY__ + "/data/gbpusd.csv")>
```

Taux de change livre/dollar Table par jours

[3]:
```
// Obtenir la date où le taux> Util.Table de change à perdu plus de 10 points␣
↪pour la livre
```

[4]:
```
let gbpusd = Gbpusd.GetSample().Rows
gbpusd
|> Seq.pairwise
|> Seq.filter (fun (before, after) -> before.GbpUsd - after.GbpUsd > 0.1M)
|> Seq.map (fun (before,_) -> before.Date.ToShortDateString())
```

[4]:
```
seq ["06/23/2016"]
```

[5]:
```
open Deedle

let titanic = Frame.ReadCsv(__SOURCE_DIRECTORY__ + "/data/titanic.csv")
```

[6]:
```
// Obtenir le taux de survivants par classe
// let byClass =
//     titanic
```

[7]:
```
let byClass =
    titanic
    |> Frame.groupRowsByString "Pclass"
    |> Frame.getCol "Survived" :> Series<(string * int),bool>
    |> Series.applyLevel fst (Series.values >> (Seq.countBy id) >> series)
    |> Frame.ofRows
    |> Frame.sortRowsByKey
    |> Frame.indexColsWith [ "Died"; "Survived" ]
```

[8]:
```
byClass?Total <- byClass?Died + byClass?Survived
```

[9]:
```
frame [ "Died (%)" => round (byClass?Died / byClass?Total * 100.0)
        "Survived (%)" => round (byClass?Survived / byClass?Total * 100.0) ]
```

[9]:
```
Deedle.Frame`2[System.String,System.String]
```

6.3 Type Provider II : Open Data

[10]:
```
open XPlot.Plotly

let wb = WorldBankData.GetDataContext()
wb
```

[10]:
```
FSharp.Data.Runtime.WorldBank.WorldBankData
```

Compilation Open data de statistiques sur le développement

[11]:
```
// Obtenir les émissions de CO2 en kt par an pour la France, l'Allemagne et les␣
↪USA
```

[12]:
```
wb.Countries.France.Indicators.``CO2 emissions (kt)``
|> Chart.Line

// wb.Countries.Germany.Indicators.``CO2 emissions (kt)``
// |> Chart.Line

// wb.Countries.``United States``.Indicators.``CO2 emissions (kt)``
// |> Chart.Line
```

[12]:
```
XPlot.Plotly.PlotlyChart
```

6.4 Type Providers III : du JSON

```
"http://api.openweathermap.org/data/2.5/forecast/daily?q=Prague&mode=json&units=metric&cnt=10&
```


```
{
  "city": {
    "id": 3067696,
    "name": "Prague",
    "coord": {
      "lon": 14.4213,
      "lat": 50.0875
    },
[...]
```


[13]:
```
type W = JsonProvider<"http://api.openweathermap.org/data/2.5/forecast/daily?
↪q=Prague&mode=json&units=metric&cnt=10&APPID=cb63a1cf33894de710a1e3a64f036a27">

let getTemps city =
  let w = W.Load("http://api.openweathermap.org/data/2.5/forecast/daily?q=" +␣
↪city +␣
↪"&mode=json&units=metric&cnt=10&APPID=cb63a1cf33894de710a1e3a64f036a27")
  [ for d in w.List -> d.Temp.Day ]
```

```
let cities = ["Paris";"Lyon";"Marseille"]
```

[14]:
```
// Afficher un barplot des températures par ville
```

[15]:
```
cities
|> Seq.map (getTemps >> Seq.indexed)
|> Chart.Column
|> Chart.WithLabels cities
```

[15]:
```
XPlot.Plotly.PlotlyChart
```

6.5 Type Providers III : du SQL

[16]:
```
[<Literal>]
let ConnectionString =
    "Data Source=" +
    __SOURCE_DIRECTORY__ + @"/data/northwindEF.db;" +
    "Version=3;foreign keys=true"

[<Literal>]
let ResolutionPath = __SOURCE_DIRECTORY__ + @"/local"

open FSharp.Data.Sql

type Sql = SqlDataProvider<
                Common.DatabaseProviderTypes.SQLITE,
                SQLiteLibrary = Common.SQLiteLibrary.SystemDataSQLite,
                ConnectionString = ConnectionString,
                ResolutionPath = ResolutionPath,
                CaseSensitivityChange = Common.CaseSensitivityChange.ORIGINAL>

let db = Sql.GetDataContext()
```

[17]:
```
// liste des noms des clients
// db.Main.Customers
// |> Seq.map etc.
```

[18]:
```
db.Main.Customers
|> Seq.map (fun c -> c.ContactName)
|> Seq.toList
```

[18]:
```
["Maria Anders"; "Ana Trujillo"; "Antonio Moreno"; "Thomas Hardy";
 "Christina Berglund"; "Hanna Moos"; "Frédérique Citeaux"; "Martín Sommer";
 "Laurence Lebihan"; "Elizabeth Lincoln"; "Victoria Ashworth";
 "Patricio Simpson"; "Francisco Chang"; "Yang Wang"; "Pedro Afonso";
 "Elizabeth Brown"; "Sven Ottlieb"; "Janine Labrune"; "Ann Devon";
 "Roland Mendel"; "Aria Cruz"; "Diego Roel"; "Martine Rancé"; "Maria Larsson";
```

```
 "Peter Franken"; "Carine Schmitt"; "Paolo Accorti"; "Lino Rodriguez";
 "Eduardo Saavedra"; "José Pedro Freyre"; "André Fonseca"; "Howard Snyder";
 "Manuel Pereira"; "Mario Pontes"; "Carlos Hernández"; "Yoshi Latimer";
 "Patricia McKenna"; "Helen Bennett"; "Philip Cramer"; "Daniel Tonini";
 "Annette Roulet"; "Yoshi Tannamuri"; "John Steel"; "Renate Messner";
 "Jaime Yorres"; "Carlos González"; "Felipe Izquierdo"; "Fran Wilson";
 "Giovanni Rovelli"; "Catherine Dewey"; "Jean Fresnière"; "Alexander Feuer";
 "Simon Crowther"; "Yvonne Moncada"; "Rene Phillips"; "Henriette Pfalzheim";
 "Marie Bertrand"; "Guillermo Fernández"; "Georg Pipps"; "Isabel de Castro";
 "Bernardo Batista"; "Lúcia Carvalho"; "Horst Kloss"; "Sergio Gutiérrez";
 "Paula Wilson"; "Maurizio Moroni"; "Janete Limeira"; "Michael Holz";
 "Alejandra Camino"; "Jonas Bergulfsen"; "Jose Pavarotti"; "Hari Kumar";
 "Jytte Petersen"; "Dominique Perrier"; "Art Braunschweiger"; "Pascale
Cartrain";
 "Liz Nixon"; "Liu Wong"; "Karin Josephs"; "Miguel Angel Paolino";
 "Anabela Domingues"; "Helvetius Nagy"; "Palle Ibsen"; "Mary Saveley";
 "Paul Henriot"; "Rita Müller"; "Pirkko Koskitalo"; "Paula Parente";
 "Karl Jablonski"; "Matti Karttunen"; "Zbyszek Piestrzeniewicz"]
```

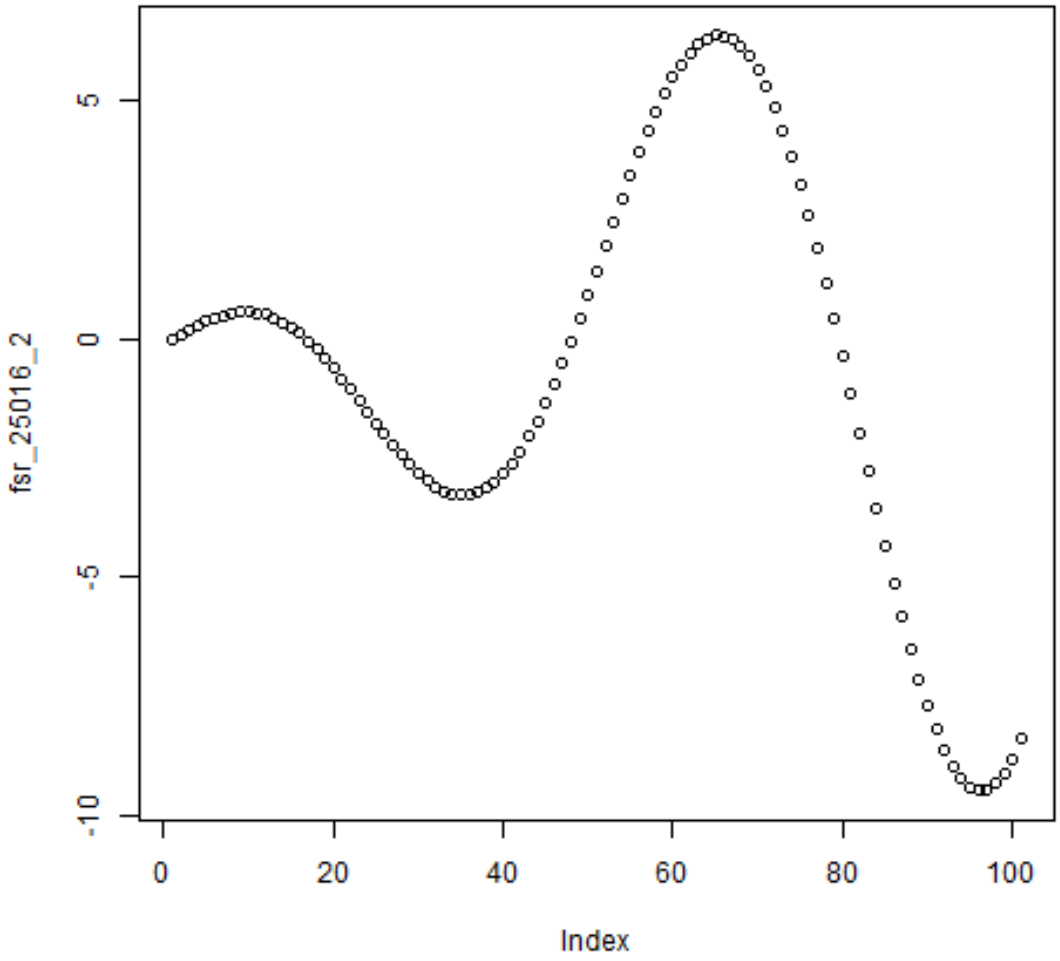
6.6 Type Provider IV : du R (!)

[19]:
```
open RDotNet
open RProvider
open RProvider.``base``
open RProvider.stats
open RProvider.graphics
open RProvider.svglite
open RProvider.grDevices
```

[20]:
```
// Petite fonction utilitaire pour afficher n'importe quel plot R dans le␣
 ↪notebook
open System.IO
let plotR (f : Lazy<SymbolicExpression>) = // f est une fonction de plot non␣
 ↪évaluée
    let path = Path.GetTempFileName()
    R.png(path) |> ignore
    f.Force() |> ignore
    R.dev_off() |> ignore
    let output = Util.Image path
    File.Delete(path)
    output
```

[21]:
```
let data = [ for x in 0. .. 0.1 .. 10. -> x * cos x ]
lazy (R.plot data) |> plotR
```

[21]:



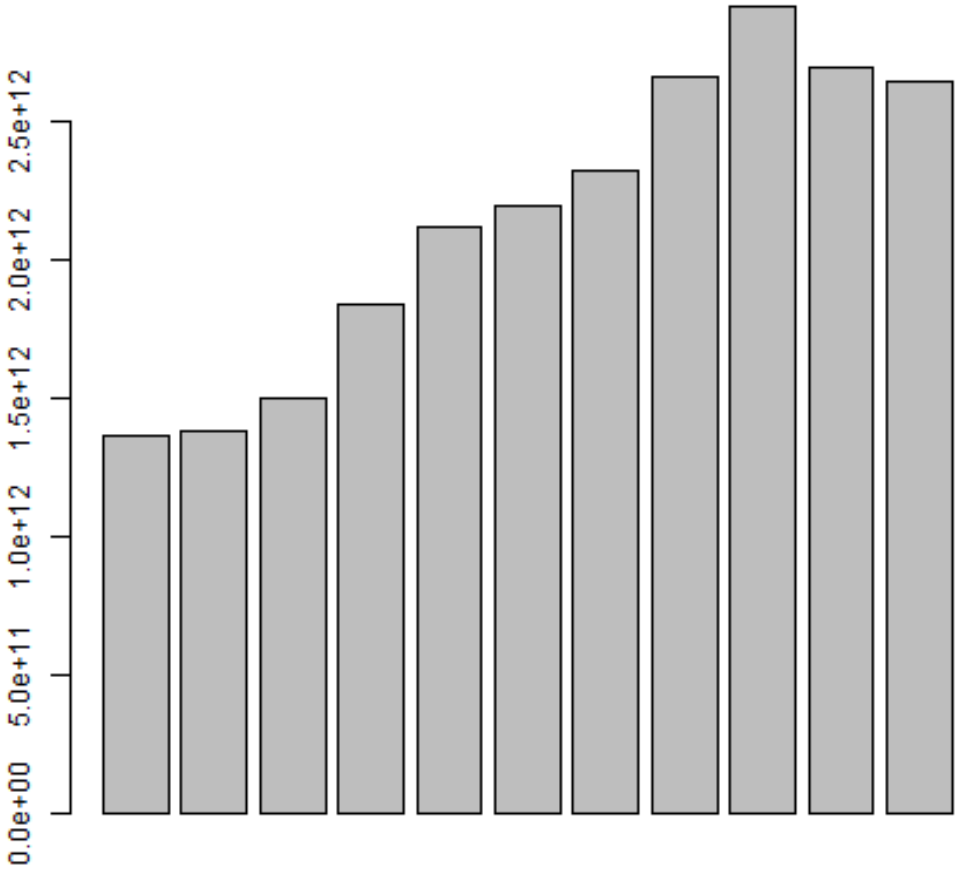

```
Microsoft R Open 3.5.2
The enhanced R distribution from Microsoft
Microsoft packages Copyright (C) 2018 Microsoft Corporation

Using the Intel MKL for parallel mathematical computing (using 4 cores).

Default CRAN mirror snapshot taken on 2019-02-01.
See: https://mran.microsoft.com/.
```

[22]:
```
lazy  ([ for y in 2000 .. 2010 -> wb.Countries.France.Indicators.``GDP (current␣
 ↪US$)``.[y] ]
      |> R.barplot)
|> plotR
```

[22]:




```
In GetStaticParametersForMethod
```

[23]:
```
let sample = query {
    for c in wb.Countries do
    where (c.Indicators.``Population, total``.[2010] > 100000000.)
```

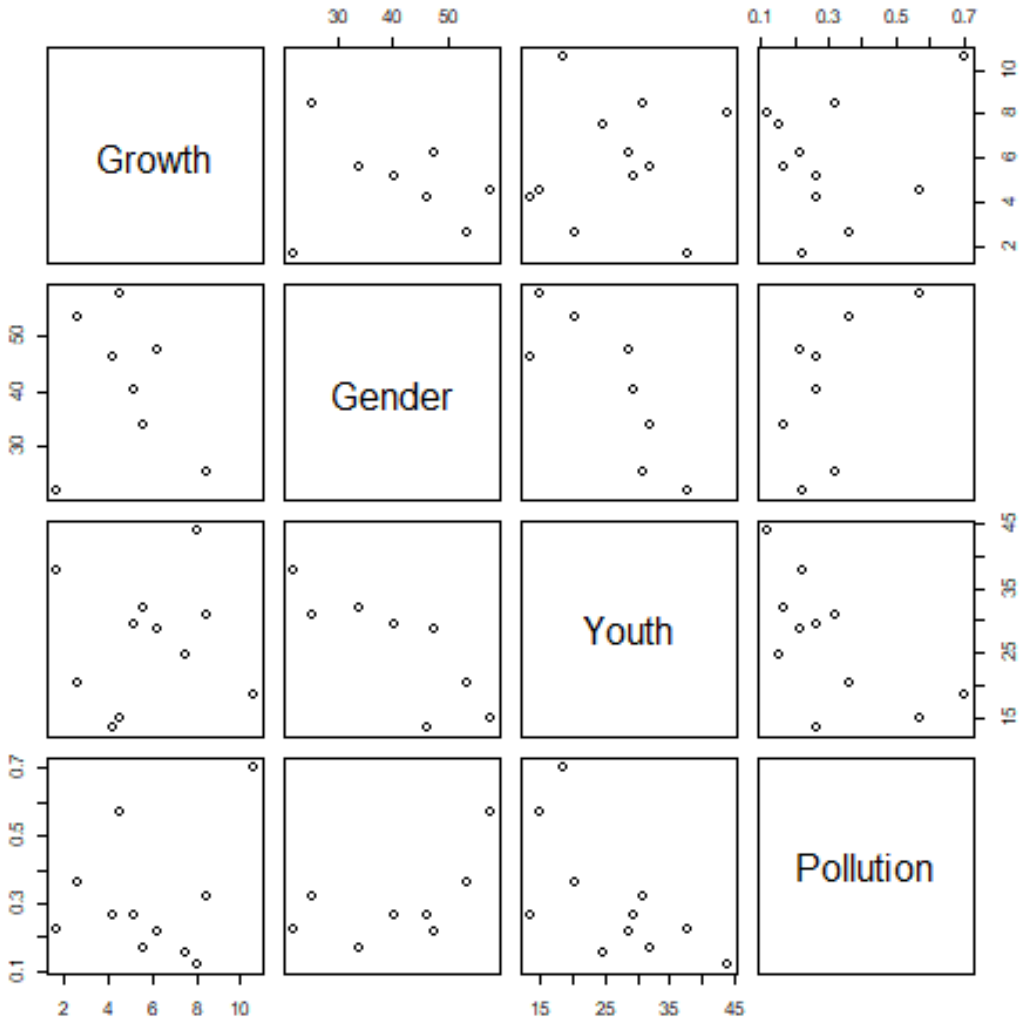
```
    select c.Indicators } |> Seq.toArray
```

[24]:
```
let growth = [ for c in sample -> c.``GDP growth (annual %)``.[2010]]
let gender = [ for c in sample -> c.``Employment to population ratio, 15+,␣
↪female (%) (national estimate)``.[2010]]
let youth = [ for c in sample -> c.``Population ages 0-14 (% of total␣
↪population)``.[2010]]
let pollution = [ for c in sample -> c.``CO2 emissions (kg per PPP $ of GDP)``.
↪[2010]]
```

[25]:
```
// Transformer ces quatres colonnes en data frame et l'afficher via R
// namedParams [
//     "Growth", growth
//     "Gender", gender
//     "Youth", youth
//     "Pollution", pollution
// ]
```

[26]:
```
lazy(namedParams [
        "Growth", growth
        "Gender", gender
        "Youth", youth
        "Pollution", pollution ]
    |> R.data_frame
    |> R.plot)
|> plotR
```

[26]:




```
In GetStaticParametersForMethod

In GetStaticParametersForMethod
```

7 Computation Expressions

Monadiquement vôtre… (Petricek and Syme, 2014)

> Les expressions de calcul F# dans fournissent une syntaxe pratique pour écrire des calculs qui peuvent être séquencés et combinés à l'aide de

constructions et de liaisons de workflow de contrôle. Selon le type d'expression de calcul, ils peuvent être considérés comme un moyen d'exprimer des monades, des monoides, des transformateurs monadiques et des foncteurs applicatifs.

Toutefois, contrairement à d'autres langages (comme la notation do-notation dans Haskell), ils ne sont pas liés à une abstraction unique et ne s'appuient pas sur des macros ou d'autres formes de surprogrammation pour accomplir une syntaxe pratique et contextuelle.

Source : https://docs.microsoft.com/fr-fr/dotnet/fsharp/language-reference/computation-express

7.1 Computation Expressions I : Des Pipes au Seq

[27]:
```
// faire une séquence infinie des entiers naturels
// let naturals = seq { 0 .. 10 }
```

[28]:
```
let naturals =
    let rec nat k =
        seq {
            yield k
            yield! nat (k + 1)
        }
    nat 0
```

[29]:
```
naturals
|> Seq.map (fun x -> x * x)
|> Seq.take 10
|> Seq.iter (printfn "%d")
```

```
0

1

4

9

16

25

36

49

64

81
```

7.2 Computation Expressions II : LINQ ou la “monade” SQL

[30]:
```
// open System.Linq
// type ComptePays = { Pays : string; Nombre : int }

// Retourner une liste des 5 pays avec le plus de clients dans l'ordre
```

[31]:
```
open System.Linq
type ComptePays = { Pays : string; Nombre : int }

// Retourner une liste des 5 pays avec le plus de clients dans l'ordre
query {
    for c in db.Main.Customers do
    groupBy c.Country into y
    sortByDescending (y.Count())
    take 5
    select { Pays = y.Key ; Nombre = y.Count() }
} |> Util.Table
```

[31]:
```
{Columns = [|"Pays"; "Nombre"|];
 Rows =
  [|[|"USA"; "13"|]; [|"France"; "11"|]; [|"Germany"; "11"|]; [|"Brazil"; "9"|];
    [|"UK"; "7"|]|];}
```

7.3 Computation Expressions III : Async

[32]:
```
open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]
```

[33]:
```
let fetchAsync(name, url:string) =
    async {
        try
            let uri = System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
```

```
    }
```

[34]:
```
// faire une fonction runAll() qui lance la récupération de ces urls
```

[35]:
```
let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()
```

```
Read 113084 characters for Bing

Read 161588 characters for Microsoft.com

Read 39685 characters for MSDN
```

7.4 Expression de calcul “custom” I : Monade Maybe

[36]:
```
type Maybe() =
    member __.Bind(p, rest) =
        match p with
        | None -> None
        | Some a -> rest a

    member __.Return(p) =
        Some p

let maybe = Maybe()
```

[37]:
```
let tryDecr x n =
  printfn "Conditionally decrementing %A by %A" x n
  if x > n then Some (x - n) else None
tryDecr
```

[37]:
```
<fun:it@4-10> : (int -> (int -> FSharpOption`1))
```

[38]:
```
let maybeDecr x = maybe {
    let! y = tryDecr x 10
    let! z = tryDecr y 30
    let! t = tryDecr z 50
    return t
```

```
  }
maybeDecr
```

[38]:
```
<fun:it@7-11> : (int -> FSharpOption`1)
```

[39]:
```
maybeDecr 100 |> printfn "%A"
maybeDecr 50  |> printfn "%A"
maybeDecr 30  |> printfn "%A"
```

```
Conditionally decrementing 100 by 10

Conditionally decrementing 90 by 30

Conditionally decrementing 60 by 50

Some 10

Conditionally decrementing 50 by 10

Conditionally decrementing 40 by 30

Conditionally decrementing 10 by 50

<null>

Conditionally decrementing 30 by 10

Conditionally decrementing 20 by 30

<null>
```

7.5 Expressions de calcul “custom” II : Eventually

- Encapsuler un calcul sous la forme d'une série d'étapes qui peuvent être évaluées une par une à la fois.
- Encoder l'état d'erreur de l'expression évaluée en cours avec un type somme

Ce code illustre plusieurs modèles typiques que vous pouvez utiliser dans vos expressions de calcul, telles que les implémentations réutilisables de certaines méthodes de générateur.

[40]:
```
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)
```

```
module Eventually =
    // The bind for the computations. Append 'func' to the
    // computation.
    let rec bind func expr =
        match expr with
        | Done value -> func value
        | NotYetDone work -> NotYetDone (fun () -> bind func (work()))

    // Return the final value wrapped in the Eventually type.
    let result value = Done value

    type OkOrException<'T> =
        | Ok of 'T
        | Exception of System.Exception

    // The catch for the computations. Stitch try/with throughout
    // the computation, and return the overall result as an OkOrException.
    let rec catch expr =
        match expr with
        | Done value -> result (Ok value)
        | NotYetDone work ->
            NotYetDone (fun () ->
                let res = try Ok(work()) with | exn -> Exception exn
                match res with
                | Ok cont -> catch cont // note, a tailcall
                | Exception exn -> result (Exception exn))

    // The delay operator.
    let delay func = NotYetDone (fun () -> func())

    // The stepping action for the computations.
    let step expr =
        match expr with
        | Done _ -> expr
        | NotYetDone func -> func ()

    // The rest of the operations are boilerplate.
    // The tryFinally operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryFinally expr compensation =
        catch (expr)
        |> bind (fun res ->
            compensation();
            match res with
            | Ok value -> result value
            | Exception exn -> raise exn)
```

```
    // The tryWith operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryWith exn handler =
        catch exn
        |> bind (function Ok value -> result value | Exception exn -> handler␣
 ↪exn)

    // The whileLoop operator.
    // This is boilerplate in terms of "result" and "bind".
    let rec whileLoop pred body =
        if pred() then body |> bind (fun _ -> whileLoop pred body)
        else result ()

    // The sequential composition operator.
    // This is boilerplate in terms of "result" and "bind".
    let combine expr1 expr2 =
        expr1 |> bind (fun () -> expr2)

    // The using operator.
    let using (resource: #System.IDisposable) func =
        tryFinally (func resource) (fun () -> resource.Dispose())

    // The forLoop operator.
    // This is boilerplate in terms of "catch", "result", and "bind".
    let forLoop (collection:seq<_>) func =
        let ie = collection.GetEnumerator()
        tryFinally
            (whileLoop
                (fun () -> ie.MoveNext())
                (delay (fun () -> let value = ie.Current in func value)))
            (fun () -> ie.Dispose())

// The builder class.
type EventuallyBuilder() =
    member __.Bind(comp, func) = Eventually.bind func comp
    member __.Return(value) = Eventually.result value
    member __.ReturnFrom(value) = value
    member __.Combine(expr1, expr2) = Eventually.combine expr1 expr2
    member __.Delay(func) = Eventually.delay func
    member __.Zero() = Eventually.result ()
    member __.TryWith(expr, handler) = Eventually.tryWith expr handler
    member __.TryFinally(expr, compensation) = Eventually.tryFinally expr␣
 ↪compensation
    member __.For(coll:seq<_>, func) = Eventually.forLoop coll func
    member __.Using(resource, expr) = Eventually.using resource expr

let eventually = EventuallyBuilder()
```

[41]:
```
// Boucle de 1 à 2 qui affiche " x = i " (pour i = 1,2) et retourne l'addition␣
↪de 3 + 4
// Essayer 1, 2 et 4 pas

let step x = Eventually.step x
```

[42]:
```
let comp = eventually {
    for x in 1..2 do
        printfn " x = %d" x
    return 3 + 4 }

// Try the remaining lines in F# interactive to see how this
// computation expression works in practice.
```

[43]:
```
// returns "NotYetDone <closure>"
comp |> step
```

[43]:
```
NotYetDone <fun:bind@11>
```

[44]:
```
// prints "x = 1"
// returns "NotYetDone <closure>"
comp |> step |> step
```

[44]:
```
NotYetDone <fun:bind@11>

 x = 1
```

[45]:
```
// prints "x = 1"
// prints "x = 2"
// returns "Done 7"
comp |> step |> step |> step |> step
```

[45]:
```
Done 7

 x = 1

 x = 2
```

8 Motifs Actifs (*Active Patterns*)

(Syme, Neverov et al., 2007)

Active patterns enable you to define named partitions that subdivide input data, so that you can use these names in a pattern matching expression just as you would for a discriminated union. You can use active patterns to decompose data in a customized manner for each partition.

Source : https://docs.microsoft.com/fr-fr/dotnet/fsharp/language-reference/active-patterns

8.1 Active Patterns I : Cas “normal”

[46]:
```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd

let TestNumber input =
   match input with
   | Even -> printfn "%d is even" input
   | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

```
7 is odd

11 is odd

32 is even
```

[47]:
```
open System.Drawing

let (|RGB|) (col : Color) =
     ( col.R, col.G, col.B )

let (|HSB|) (col : Color) =
   ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let printRGB (col: Color) =
   match col with
   | RGB(r, g, b) -> printfn " Red: %d Green: %d Blue: %d" r g b

let printHSB (col: Color) =
   match col with
   | HSB(h, s, b) -> printfn " Hue: %f Saturation: %f Brightness: %f" h s b

let printAll col colorString =
  printfn "%s" colorString
```

```
  printRGB col
  printHSB col

printAll Color.Red "Red"
printAll Color.Black "Black"
printAll Color.White "White"
printAll Color.Gray "Gray"
printAll Color.BlanchedAlmond "BlanchedAlmond"
```

```
Red

 Red: 255 Green: 0 Blue: 0

 Hue: 0.000000 Saturation: 1.000000 Brightness: 0.500000

Black

 Red: 0 Green: 0 Blue: 0

 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.000000

White

 Red: 255 Green: 255 Blue: 255

 Hue: 0.000000 Saturation: 0.000000 Brightness: 1.000000

Gray

 Red: 128 Green: 128 Blue: 128

 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.501961

BlanchedAlmond

 Red: 255 Green: 235 Blue: 205

 Hue: 36.000000 Saturation: 1.000000 Brightness: 0.901961
```

8.2 Active Patterns II : partiel

[48]:
```
let (|Integer|_|) (str: string) =
   let mutable intvalue = 0
   if System.Int32.TryParse(str, &intvalue) then Some(intvalue)
   else None

let (|Float|_|) (str: string) =
   let mutable floatvalue = 0.0
   if System.Double.TryParse(str, &floatvalue) then Some(floatvalue)
   else None

let parseNumeric str =
   match str with
     | Integer i -> printfn "%d : Integer" i
     | Float f -> printfn "%f : Floating point" f
     | _ -> printfn "%s : Not matched." str

parseNumeric "1.1"
parseNumeric "0"
parseNumeric "0.0"
parseNumeric "10"
parseNumeric "Something else"
```

```
1.100000 : Floating point

0 : Integer

0.000000 : Floating point

10 : Integer

Something else : Not matched.
```

Sometimes, you need to partition only part of the input space. In that case, you write a set of partial patterns each of which match some inputs but fail to match other inputs. Active patterns that do not always produce a value are called partial active patterns; they have a return value that is an option type. To define a partial active pattern, you use a wildcard character (_) at the end of the list of patterns inside the banana clips. The following code illustrates the use of a partial active pattern.

8.3 Active Patterns III : paramétrisé

[49]:
```
open System.Text.RegularExpressions

// ParseRegex parses a regular expression and returns a list of the strings␣
 ↪that match each group in
// the regular expression.
// List.tail is called to eliminate the first element in the list, which is the␣
 ↪full matched expression,
// since only the matches for each group are wanted.
let (|ParseRegex|_|) regex str =
   let m = Regex(regex).Match(str)
   if m.Success
   then Some (List.tail [ for x in m.Groups -> x.Value ])
   else None
```

Active patterns always take at least one argument for the item being matched, but they may take additional arguments as well, in which case the name parameterized active pattern applies. Additional arguments allow a general pattern to be specialized. For example, active patterns that use regular expressions to parse strings often include the regular expression as an extra parameter, as in the following code, which also uses the partial active pattern Integer defined in the previous code example. In this example, strings that use regular expressions for various date formats are given to customize the general ParseRegex active pattern. The Integer active pattern is used to convert the matched strings into integers that can be passed to the DateTime constructor.

[50]:
```
// Three different date formats are demonstrated here. The first matches two-
// digit dates and the second matches full dates. This code assumes that if a␣
 ↪two-digit
// date is provided, it is an abbreviation, not a year in the first century.
let parseDate str =
   match str with
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{1,2})$" [Integer m; Integer d;␣
 ↪Integer y]
         -> new System.DateTime(y + 2000, m, d)
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{3,4})" [Integer m; Integer d;␣
 ↪Integer y]
         -> new System.DateTime(y, m, d)
     | ParseRegex "(\d{1,4})-(\d{1,2})-(\d{1,2})" [Integer y; Integer m;␣
 ↪Integer d]
         -> new System.DateTime(y, m, d)
     | _ -> new System.DateTime()
```

[51]:
```
let dt1 = parseDate "12/22/08"
let dt2 = parseDate "1/1/2009"
let dt3 = parseDate "2008-1-15"
```

```
let dt4 = parseDate "1995-12-28"

printfn "%s %s %s %s" (dt1.ToString()) (dt2.ToString()) (dt3.ToString()) (dt4.
 ↪ToString())
```

```
12/22/2008 00:00:00 01/01/2009 00:00:00 01/15/2008 00:00:00 12/28/1995 00:00:00
```

9 Technologies Web : Développement “Full Stack”

- SAFE Stack
- WebSharper

9.1 Fable : un transpileur JS “état de l’art”

(Nunez and Fahad, 2016)

- Programmation fonctionnelle + typage fort
- Sortie JS ``propre'' et derniers standard (ES2015+)
- Interaction facile avec tout l'écosystème JS (npm)
- Pratiquement tout le coeur F#, ``pruning'' efficace sur le code à la compilation

9.2 SAFE-Stack

- Programmation quasi ``isomorphique'' Client-Serveur
- Basée sur le modèle/vue/mise à jour popularisé par elm
- Model --- the state of your application
- Update --- a way to update your state
- View --- a way to view your state as HTML
- FRP avec React (support HMR etc…)

Hot Module Replacement (HMR) allows to update the UI of an application while it is running, without a full reload. In SAFE stack apps, this can dramatically speed up the development for web and mobile GUIs, since there is no need to ``stop'' and ``reload'' and application. Instead, you can make changes to your views and have them immediately update in the browser, without the need to restart the application.

10 References

(Syme, 2019) Syme Don, ``*The Early History of F# (HOPL IV-second draft)*'', , vol. , number , pp. , 2019. online

(Kennedy and Syme, 2001) A. Kennedy and D. Syme, ``*Design and implementation of generics for the. net common language runtime*'', ACM SigPlan Notices, 2001.

(Syme, Battocchi et al., 2012) Syme Don, Battocchi Keith, Takeda Kenji et al., ``*Strongly-typed language support for internet-scale information sources*'', Technical Report MSR-TR-2012--101, Microsoft Research, vol. , number , pp. , 2012.

(Petricek and Syme, 2014) T. Petricek and D. Syme, ``*The F# computation expression zoo*'', International Symposium on Practical Aspects of Declarative Languages, 2014.

(Syme, Neverov et al., 2007) D. Syme, G. Neverov and J. Margetson, ``*Extensible pattern matching via a lightweight language extension*'', ACM SIGPLAN Notices, 2007.

(Nunez and Fahad, 2016) Alfonso Garcia-Caro Nunez and Suhaib Fahad, ``*Mastering F#*'', 2016.