Some F# for the Erlang programmer

Don Syme, F# Team, Microsoft, http://fsharp.net
We’ve been busy 😊
F# Topics for the Erlang Programmer

- A little about F#
- Basics
- Syntax
- Types
- Pattern Matching
- Objects
- In-memory Agent Programming
Simplicity: Scripting

F#

open System

let greeting = "hello"

Console.WriteLine(greeting)

C#

using System;

namespace ConsoleApplication1
{
    class Program
    {
        static string greeting()
        {
            return "hello";
        }
        static void Main(string[] args)
        {
            Console.WriteLine(greeting);
        }
    }
}
Simplicity: Functional Data

```csharp
Tuple<U,T> Swap<T,U>(Tuple<T,U> t)
{
    return new Tuple<U,T>(t.Item2, t.Item1)
}

ReadOnlyCollection<Tuple<T,T,T>> Rotations<T>(Tuple<T,T,T> t)
{
    new ReadOnlyCollection<int>
    (new Tuple<T,T,T>[] { new Tuple<T,T,T>(t.Item1,t.Item2,t.Item3);
        new Tuple<T,T,T>(t.Item3,t.Item1,t.Item2);
        new Tuple<T,T,T>(t.Item2,t.Item3,t.Item1);
    });
}

int Reduce<T>(Func<T,int> f,Tuple<T,T,T> t)
{
    return f(t.Item1) + f(t.Item2) + f(t.Item3);
}
```
Simplicity: Functional Data

type Event =
| Price of float
| Split of float
| Dividend of float<money>

C#

```csharp
public abstract class Event { }
public abstract class PriceEvent : Event
{
    public Price Price { get; private set; }
    public PriceEvent(Price price)
    {
        this.Price = price;
    }
}
public abstract class SplitExpr : Event
{
    public double Factor { get; private set; }
    public SplitExpr(double factor)
    {
        this.Factor = factor;
    }
}
public class DividendEvent : Event { }
...
```

F#
Simplicity: Functional Parallel

Async.Parallel [ http "www.google.com";
http "www.bing.com";
http "www.yahoo.com"; ]

|> Async.RunSynchronously
F#: Influences

OCaml

F#

Similar core language

Similar object model

C#/.NET
I've been coding in F# lately, for a production task.

F# allows you to move smoothly in your programming style... I start with pure functional code, shift slightly towards an object-oriented style, and in production code, I sometimes have to do some imperative programming.

I can start with a pure idea, and still finish my project with realistic code. You're never disappointed in any phase of the project!

Erlang & F# - Overview
Erlang/F# - Aims are Different

**F#**: Practical Typed Functional Programming
- “Interop, interop, interop”
- “Objects are the foundation of .NET libraries. They are very effective. Embrace them.”
- “Parallel, multi-core, asynchronous”
- “Nice, practical functional features”
- Native-code, C# performance, often near C++

**Erlang**: Practical Concurrent Actors
- Telephony, now general I/O parallel programming
F# & Erlang - The Familiar

Technically, both are
- Functional core
- Strict
- Single Assignment
- Pattern Matching
- Strong emphasis on Tuples, Lists, Recursion

Or... the two main practical languages where it’s important to know what a tailcall is 😊
F# & Erlang - The Familiar

Methodologically, both use
- Design by prototyping
- Exploration & experimentation
- Good for smart, programmer/architects

Historically
- There is a lot of overlap, shared vocabulary and knowledge
However...

- F# is not a functional-only language
  - Imperative state, arrays
  - Objects
  - Meta-programming
  - .NET libraries

- F# does things differently
  - Syntax
  - Types
  - Libraries
  - Concurrency via an “async { ... }” modality
  - + much inherited from .NET
F# Topics
F# Topics for the Erlang Programmer

- Basics
- Syntax
- Types
- Pattern Matching
- Objects
- In-memory Agent Programming
Clones

Erlang has inspired several clones of its concurrency facilities for other languages:

- C#: Retlang
- F#: MailboxProcessor
- JavaScript: Er.js
- Lisp: erlang-in-lisp, CL-MUPROC, Distel
- .NET: Async.Fx, simulating a joint from Microsoft Research
Tutorial: Fundamentals
```ml
let computeDerivative f x =
    let p1 = f (x - 0.05)
    let p2 = f (x + 0.05)
    (p2 - p1) / 0.1
```
let computeDerivative f x =
  let p1 = f (x - 0.05)

  let p2 = f (x + 0.05)

  (p2 - p1) / 0.1
Basics

- module(fact).
- export([fac/1]).

fac(0) -> 1
fac(N) -> N * fac(N-1).

module Fact

let rec fac x =
  match x with
  | 0 -> 1
  | n -> n * fac (n-1)
Erlang

-module(fact).
-export([fac/1]).
fac(0) -> 1
fac(N) -> N * fac(N-1).

F#

module Fact
let rec fac = function
| 0    -> 1
| n    -> n * fac (n-1)
Basics

```fsharp
let rec quicksort = function
  | [] -> []
  | pivot::rest ->
    quicksort (List.filter (fun x -> x < pivot) rest) @
    [pivot] @
    quicksort (List.filter (fun x -> x >= pivot) rest)
```

Basics

F#

let rec quicksort = function
| [] -> []
| pivot::rest ->
  quicksort (rest |> List.filter (fun x -> x < pivot)) @
  [pivot] @
  quicksort (rest |> List.filter (fun x -> x >= pivot))
**Basics**

```
let rec quicksort = function
| [] -> []
| pivot::rest ->
  let lo,hi = rest |> List.partition (fun x -> x < pivot)
  quicksort lo @ [rest] @ quicksort hi
```
Functional– Pipelines

The pipeline operator

\[ x \mid> f \]
Functional– Pipelines

Successive stages in a pipeline

\[ x \xrightarrow{f_1} x \xrightarrow{f_2} x \xrightarrow{f_3} \]
let squares (n, m) = 
    [ for x in n .. m do 
      yield (x, x*x) ]

let activities children = 
  [ for child in children do 
    yield "WakeUp.fs"
    yield! morningActivities child ]
let rec allFiles dir = [ for file in Directory.GetFiles dir do yield file 
    for sub in Directory.GetDirectories dir do yield! allFiles sub ]

allFiles @"C:\LogFiles"

We can do I/O here
This is F#, not Haskell
let rec allFiles dir = seq {
  for file in Directory.GetFiles dir do yield file
  for sub in Directory.GetDirectories dir do yield! allFiles sub
}

allFiles @"C:\LogFiles"
 |> Seq.take 100
 |> show
Generating Data with seq { ... }

```plaintext
let rec randomWalk x = 
    seq { yield x
          yield! randomWalk (x+rnd()) }
```

On-demand, infinite
Generating Data with seq { ... }

```
[ 0..1000 ]
[ for x in 0..1000 -> (x, x * x) ]
[| for x in 0..1000 -> (x, x * x) |]
seq { for x in 0..1000 -> (x, x * x) }
```
Generating Structured Data

type Suit =
    | Heart
    | Diamond
    | Spade
    | Club

Union type (no data = enum)

type PlayingCard =
    | Ace of Suit
    | King of Suit
    | Queen of Suit
    | Jack of Suit
    | ValueCard of int * Suit

Union type with data
let suits = [ Heart; Diamond; Spade; Club ]

let deckOfCards =
[ for suit in suits do yield
  Ace suit
  King suit
  Queen suit
  Jack suit
  for value in 2 .. 10 do
    yield ValueCard (value, suit) ]

Generate a deck of cards
Tutorial: Actors & Async
The actor model consists of a few key principles:

- No shared state
- Lightweight processes
- Asynchronous message-passing
- Mailboxes to buffer incoming messages
- Mailbox processing with pattern matching
Actors

Erlang

```erlang
code() ->
  spawn(echo, loop, []).

loop() ->
  receive {From, Message} -> From ! Message, loop()
  end.
```

F#

```fsharp
let (a:Agent<_>) => x = a.Post x

let agent =
  Agent.Start(fun inbox ->
    async { while true do
      let! (from, msg) = inbox.Receive()
      from <- msg }
```
Actors

Erlang

\begin{verbatim}
start() ->
  spawn(echo, loop, []).

loop() ->
  receive {From, Message} -> From ! Message, loop()
  end.
\end{verbatim}

\begin{verbatim}
let ((<->) (a:Agent<>)) x = a.Post x

let rec loop (inbox:Agent<>)) =
  async {
    let! (from, msg) = inbox.Receive()
    from.Post msg
    return! loop inbox
  }

let agent = Agent.Start(loop)
\end{verbatim}
So what is async { ... }?
F# is a **Parallel** Language
(Multiple active **computations**)

F# is a **Reactive** Language
(Multiple pending **reactions**)

- GUI Event
- Page Load
- Timer Callback
- Query Response
- HTTP Response
- Web Service Response
- Disk I/O Completion
- Agent Gets Message
Computational Model Example

- (e.g. Click)
- Single Threaded GUI
- Handlers (callbacks)
- Pending Reactions
  (e.g. I/O response)
async { let! res = <async-event>  
  ...  
}
async { ... }

```csharp
async { let! image = ReadAsync "cat.jpg"
    let image2 = f image
    do! WriteAsync image2 "dog.jpg"
    do printfn "done!"
    return image2 }
```

- Asynchronous "non-blocking" action
- Continuation/Event callback
The many uses of F# async { ... }

- Sequencing I/O requests

```fsharp
async { let! lang = detectLanguageAsync text
          let! text2 = translateAsync (lang,"da",text)
          return text2 }
```

- Sequencing CPU computations and I/O requests

```fsharp
async { let! lang = detectLanguageAsync text
          let! text2 = translateAsync (lang,"da",text)
          let text3 = postProcess text2
          return text3 }
```
The many uses of F# async { ... }

- Parallel CPU computations

```fsharp
Async.Parallel [ async { return (fib 39) }; async { return (fib 40) }; ]
```

- Parallel I/O requests

```fsharp
Async.Parallel
    [ for target in langs -> translateAsync (lang,target,text) ]
```
Demo: Web Crawling
ready to learn some parallel I/O programming?

Translating...
en --> ar: "مرحبا، تونأ جاهأا لتترع م ع بعض"
en --> bg: "Здравейте са сте готови да научат някои паралелно в/И програмиране?"
en --> zh-CHS: "您好, 你准备好要学习一些并行 I/O 编程吗？"
en --> zh-CHT: "您好, 您準備好要學習一些並行 I/O 程式設計嗎？"
en --> cs: "Ahoj jsou vám připraveni učit několik paralelních I/O programování?"
en --> da: "Hej, er du klar til at lære nogle parallelt I/O programmering?"
en --> nl: "Hello, worden u klaar voor meer informatie over sommige parallelle I/O programmeren?"
en --> en: "Hello, are you ready to learn some parallel I/O programming?"
en --> fr: "Bonjour, s'ou ki pare pou yo aprann kèk paralèl I/O pwogramasyon ?"
F# async Parallel Lightweight Parallel Request F# async Parallel Request Handlers Lightweight Parallel Agents
F# async

Parallel

Server
F# example: Serving 5,000+ simultaneous TCP connections with ~10 threads

```fsharp
/// Write a stream of requests to a server
let handleServerRequest (client: TcpClient) =
    async {
        use stream = client.GetStream()

        // Write header
        do! stream.AsyncWrite(header)

        while true do
            // Write one quote
            do! stream.AsyncWrite(quote())
            // Wait for the next quote
            do! Async.Sleep ioWaitPerQuote
    }

let server() =
    AsyncTcpServer(IPAddressLOOPBACK, 10000, handleServerRequestAsync)
```
F# async

Parallel

Server

Lightweight Parallel Agents
F# async
Parallel
Server
Agents
let agent =

Agent.Start(fun inbox ->
    async { while true do
        let! msg = inbox.Receive()
        printfn "got message %s" msg
    }
)

agent.Post "three"
agent.Post "four"
let agents =
    [ for i in 1 .. 100000 ->
      Agent.Start(fun inbox ->
        async {
          while true do
            let! msg = inbox.Receive()
            printfn "%d got message %s" i msg })]

for agent in agents do
  agent.Post "hello"
Agents – Typed messages

type Message =
    | Message1
    | Message2 of int * string

let agent =
    Agent.Start(fun inbox ->
        async {
            while true do
                let! msg = inbox.Receive()
                match msg with
                    | Message1 -> ...  
                    | Message2 (x,y) -> ...  
        })

agent.Post Message1
agent.Post (Message2(3,"t"))
Agents – Untyped messages

```ml
type Message = obj

let agent = Agent.Start(fun inbox -> async { while true do let! msg = inbox.Receive() match msg with | :? string as s -> ... | :? int as d -> ... | _ -> ... })

agent.Post 3
agent.Post "three"
```
Agents – State Isolation

```
let agents =
    Agent.Start(fun inbox ->
    async {
        let state = Dictionary<int,string>()
        while true do
            let! key,value = inbox.Receive()
            state.[key] <- value
        }
    }

for i in 0..10000 do
    agent.Post (i, string i)
```
let agents = 
[ for i in 0 .. 100000 -> 
  Agent.Start(fun inbox ->
    async { while true do
      let! a,b,reply = inbox.Receive()
      msg <-- (a+b) }}] 
Async.Parallel 
[ for agent in agents -> 
  agent.PostAndAsyncReply (fun r -> (10,10,r)) ]
Miscellaneous Actor Topics

- **Error Orchestration**
  - Add listener to “Error” event on agents

- **Isolation**
  - Isolated state in agents is easy

- **Scanning**
  - Inbox.Scan

- **Timeouts**
  - timeout=10, inbox.TryReceive, inbox.TryScan, inbox.DefaultTimeout

- **Scheduling**
  - Rarely tweaked. Set the global “Synchronization Context” or give high-priority agents their own thread.
Tutorial: Objects
Objects

Class Types

type ObjectType(args) =

  let internalValue = expr
  let internalFunction args = expr
  let mutable internalState = expr

  member x.Prop1 = expr
  member x.Meth2 args = expr

Interface Types

type IObject =
  interface ISimpleObject
  abstract Prop1 : type
  abstract Meth2 : type -> type

Constructing Objects

new FileInfo("c:\misc\test.fs")
type Vector2D (dx:double, dy:double) =

let d2 = dx*dx+dy*dy

member v.DX = dx

member v.DY = dy

member v.Length = sqrt d2

member v.Scale(k) = Vector2D (dx*k,dy*k)
type HuffmanEncoding(freq: seq<char*int>) =

...<50 lines of beautiful functional code>...

member x.Encode(input: seq<char>) =
    encode(input)

member x.Decode(input: seq<char>) =
    decode(input)
type MutableVector2D (dx:double, dy:double) =

    let mutable currDX = dx
    let mutable currDY = dy

    member v.DX = currDX
    member v.DY = currDY

    member v.MoveX x = currDX <- currDX + x
    member v.MoveY y = currDY <- currDY + y
OO: F#/C# comparisons

Source: F# Object-Oriented Quick Guide
Class with Properties

```csharp
// Usage:
let v = Vector(10., 10.)
let x = v.X
let y = v.Y

public class Vector
{
    double x;
    double y;

    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double X
    {
        get { return this.x; }
    }
    public double Y
    {
        get { return this.y; }
    }
}
```

```csharp
// Usage:
Vector v = new Vector(10, 10);
double x = v.X;
double y = v.Y;
```
Wrapping Up
In Summary

Simple, powerful, and productive

A powerful addition to .NET/Visual Studio

F# + .NET 4.0 greatly simplify parallelism

F# is ready for use in production with VS2010
Not Covered

- Many minor topics
- Distribution
  - F# uses .NET for distributed computing
  - Strong emphasis on SOA (Web/REST Services)
  - “Windows Communication Foundation”
  - Cloud (Azure 2010)
- Hot swapping
  - .NET does not support “hot code swapping” in normal use. Some can be architected in.
Latest Books about F#

www.fsharp.net
Questions & Discussion

http://fsharp.net
http://blogs.msdn.com/dsyme
http://meetup.com/FSharpLondon