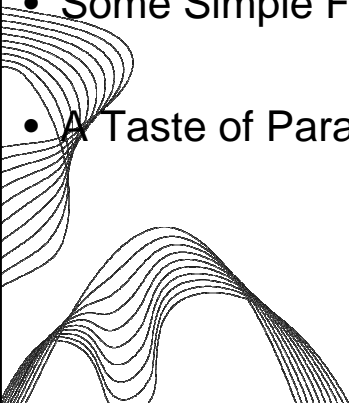


# F#

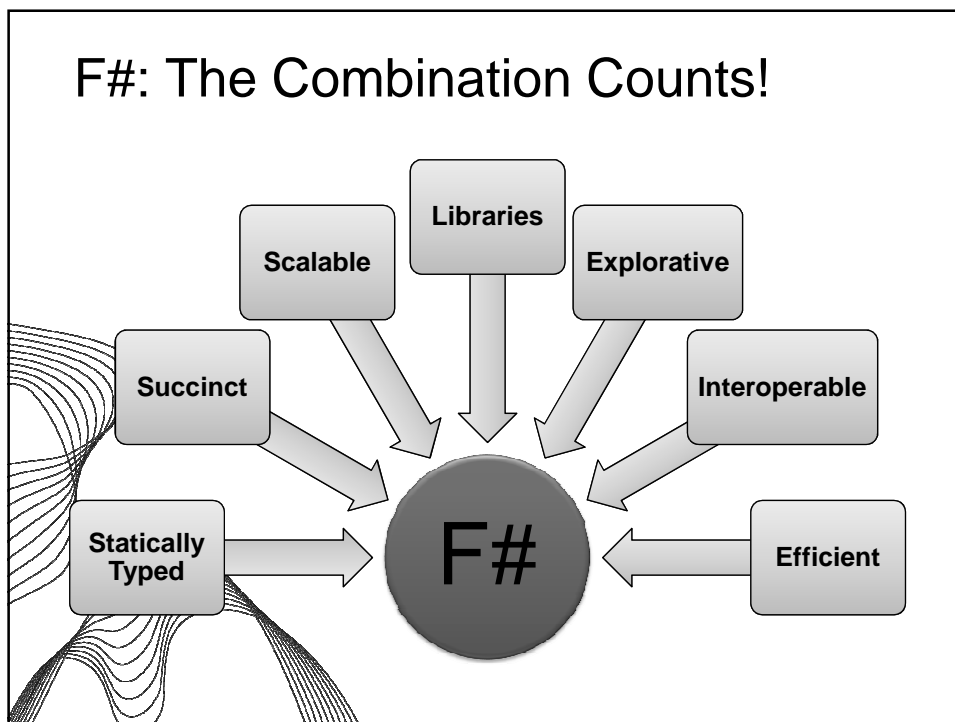
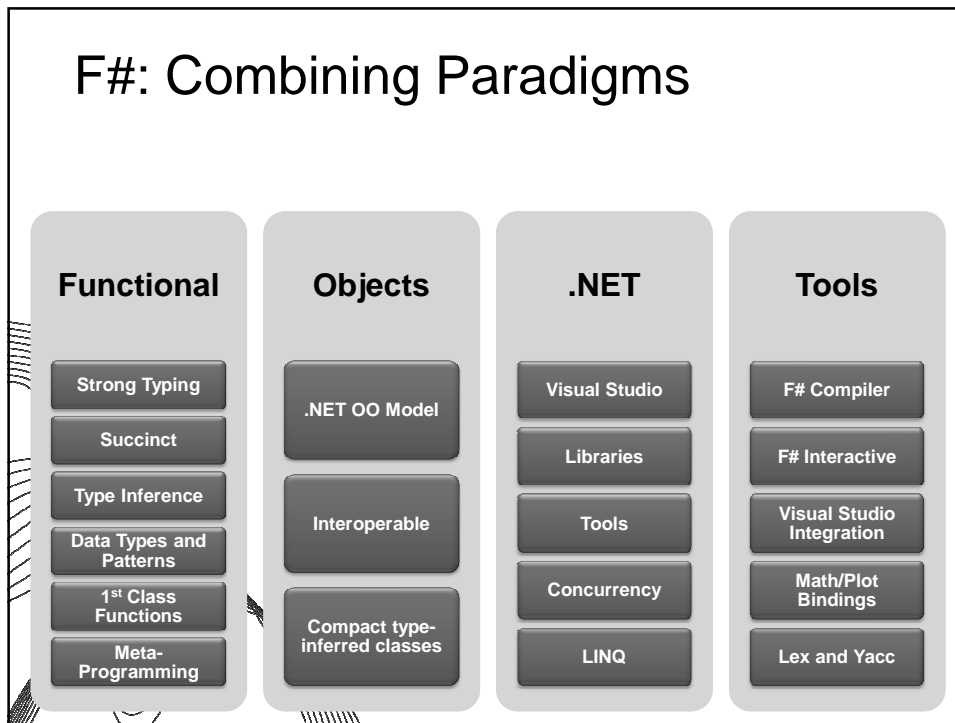
Succinct, Expressive, Efficient  
Functional Programming for .NET

The F# Team  
Microsoft Developer Division, Redmond  
Microsoft Research, Cambridge



## Topics

- What is F# about?
- Some Simple F# Programming
- A Taste of Parallel/Reactive with F#



## F#: Combining Paradigms

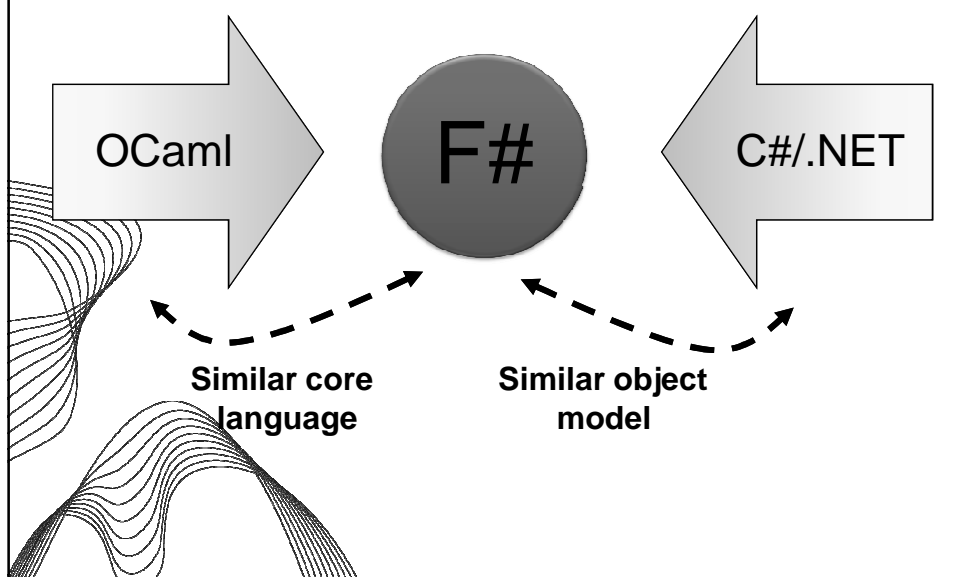
*I've been coding in F# lately, for a production task.*

*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!*

Julien Laugel, Chief Software Architect, [www.eurostocks.com](http://www.eurostocks.com)

## F#: Influences



## The Path to Mastering F#

Topic	Covered Today
Scoping and "let"	✓
Tuples	✓
Pattern Matching	✓
Working with Functions	✓
Sequences, Lists, Options	✓
Records and Unions	✓
Basic Imperative Programming	✓
Basic Objects and Types	✓
The F# Libraries	✗
Advanced Functional/Imperative	✗
Advanced Functional/OO	✗
Language Oriented Programming	✓ (later)
Parallel and Asynchronous	✓ (later)

## Quick Tour

### *Comments*

```
// comment
```

```
(* comment *)
```

```
/// XML doc comment
```

```
let x = 1
```

## Quick Tour

### Overloaded Arithmetic

<code>x + y</code>	Addition
<code>x - y</code>	Subtraction
<code>x * y</code>	Multiplication
<code>x / y</code>	Division
<code>x % y</code>	Remainder/modulus
<code>-x</code>	Unary negation

### Booleans

<code>not expr</code>	Boolean negation
<code>expr &amp;&amp; expr</code>	Boolean "and"
<code>expr    expr</code>	Boolean "or"

## Quick Tour: Types

### Basic Types and Literals

<code>sbyte</code>	= <code>System.SByte</code>	<code>76y</code>
<code>byte</code>	= <code>System.Byte</code>	<code>76uy</code>
<code>int16</code>	= <code>System.Int16</code>	<code>76s</code>
<code>uint16</code>	= <code>System.UInt16</code>	<code>76us</code>
<code>int32</code>	= <code>System.Int32</code>	<code>76</code>
<code>uint32</code>	= <code>System.UInt32</code>	<code>76u</code>
<code>int64</code>	= <code>System.Int64</code>	<code>76L</code>
<code>uint64</code>	= <code>System.UInt64</code>	<code>76UL</code>
<code>string</code>	= <code>System.String</code>	<code>"abc", @"c:\etc"</code>
<code>single</code>	= <code>System.Single</code>	<code>3.14f</code>
<code>double</code>	= <code>System.Double</code>	<code>3.14, 3.2e5</code>
<code>char</code>	= <code>System.Char</code>	<code>'7'</code>
<code>nativeint</code>	= <code>System.IntPtr</code>	<code>76n</code>
<code>unativeint</code>	= <code>System.UIntPtr</code>	<code>76un</code>
<code>bool</code>	= <code>System.Boolean</code>	<code>true, false</code>
<code>unit</code>	= <code>Microsoft.FSharp.Core.Unit</code>	<code>()</code>

### Basic Type Abbreviations

<code>int8</code>	= <code>sbyte</code>
<code>uint8</code>	= <code>byte</code>
<code>int</code>	= <code>int32</code>
<code>float32</code>	= <code>single</code>
<code>float</code>	= <code>double</code>

## Orthogonal & Unified Constructs

- Let “let” simplify your life...

Type inference. The safety of C# with the succinctness of a scripting language

Bind a static value

```
let data = (1,2,3)
```

Bind a static function

```
let f(a,b,c) =  
    let sum = a + b + c  
    let g(x) = sum + x*x  
    g(a), g(b), g(c)
```

Bind a local value

Bind a local function

Demo: Let's WebCrawl...

## Orthogonal & Unified Constructs

- Functions: like delegates + unified and simple

One simple mechanism, many uses

predicate = 'a -> bool

send = 'a -> unit

threadStart = unit -> unit

comparer = 'a -> 'a -> int

hasher = 'a -> int

equality = 'a -> 'a -> bool

```
(fun x -> x + 1)
let f(x) = x + 1
(f,f)
val f : int -> int
```

Declare a function

A pair of functions

A function type

## F# - Functional

```
let f x = x+1
let pair x = (x,x)
let fst (x,y) = x
let data = (Some [1;2;3], Some [4;5;6])
match data with
| Some(nums1), Some(nums2) -> nums1 @ nums2
| None, Some(nums) -> nums
| Some(nums), None -> nums
| None, None -> failwith "missing!"
```

## F# - Functional

List.map            Seq.fold  
 Array.filter    Lazy    Range    Expressions    Set.union

Map            LazyList    Events    Async...  
 List via query

```

0..1000 ]
[ for x in 0..10 -> (x, x * x) ]
[ | for x in 0..10 -> (x, x * x) | ]
seq { for x in 0..10 -> (x, x * x) }
    
```

Array via query

IEnumerable via query

## F# - Functional + Queries

```

SQL <@ { for customer in db.Customers do
        for employee in db.Employees do
            if customer.Name = employee.Name then
                yield (customer.Name, employee.Address) } @>
    
```

SQL : Expr<seq<'a>> -> seq<'a>

LINQ SQLMetal

SQL Database



## Immutability the norm...

```

//-----
// Part 1. Adjust some constants

let PI = 3.141592654

PI <- 4.0
This value is not mutable.

type Person =
{ Name : string;
  Birth: DateTime }

let bob =
{ Name = "bob";
  Birth = DateTime(15,8,1980) }

// OK
let bobJunior =
{ bob with Birth = DateTime(23,5,2006) }

// Not OK!
bob.Birth <- DateTime(23,5,2006)

```

**Data is immutable by default**

**Values may not be changed**

**Copy & Update**

**Not Mutate**

Description	File	Line	Column
1 error FS0005: This field is not mutable	test.fs	18	1

## In Praise of Immutability

- Immutable objects can be relied upon
- Immutable objects can transfer between threads
- Immutable objects can be aliased safely
- Immutable objects lead to (different) optimization opportunities

## F# - Imperative + Functional

Using .NET  
collections

```
open System.Collections.Generic

let dict = new Dictionary<int,string>(1000)

dict.[17] <- "Seventeen"
dict.[1000] <- "One Grand"

for (KeyValue(k,v)) in dict do
    printfn "key = %d, value = %s" k v
```

## F# - Imperative + Functional

"use" =  
C# "using"

```
open System.IO
open System.Collections.Generic

let readAllLines(file) =
    use inp = File.OpenText file
    let res = new List<_>()
    while not(inp.EndOfStream) do
        res.Add(inp.ReadLine())
    res.ToArray()
```


## F# - Sequences

Sequence  
Expressions and  
On-demand I/O

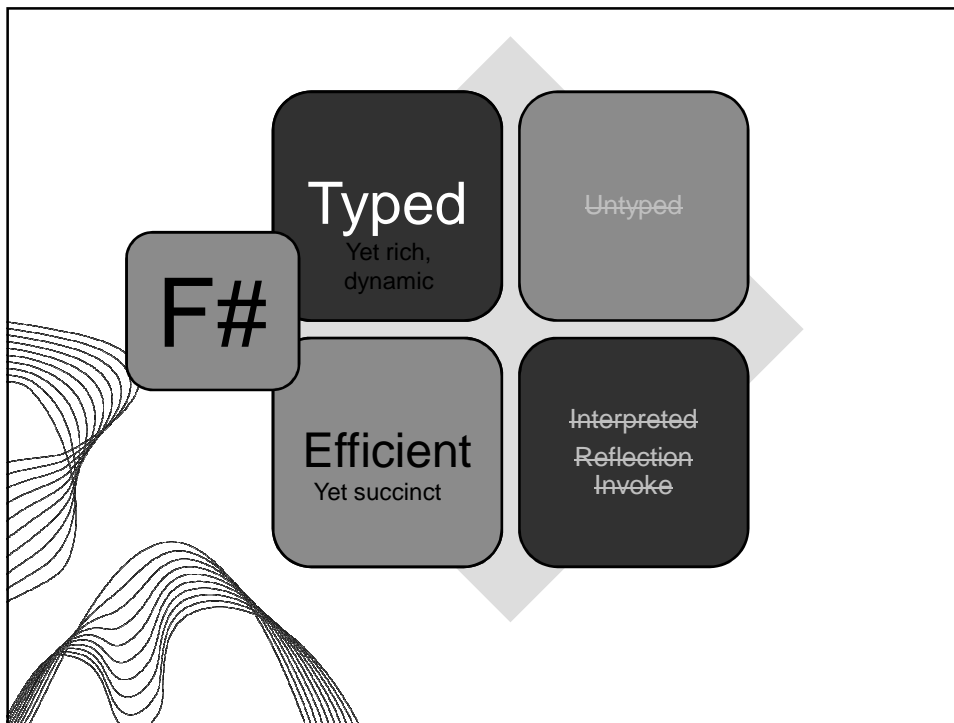
```
open System.IO
let rec allFiles(dir) =
    seq
    { for file in Directory.GetFiles(dir) do
      yield file
    for sub in Directory.GetDirectories(dir) do
      yield! allFiles(sub) }

allFiles(@"C:\WINDOWS") |> Seq.take 100 |> show
```

## Weakly Typed? Slow?



Looks Weakly typed?  
Maybe Dynamic?



## F# - Imperative + Functional

```

open System.IO
let allLines =
    seq { use inp = File.OpenText "test.txt"
          while not(inp.EndOfStream) do
            yield (inp.ReadLine()) }

allLines
    |> Seq.truncate 1000
    |> Seq.map (fun s -> uppercase s,s)
    |> Seq.to_array
  
```

Read lines on demand

Pipelines

## F# - Objects + Functional

```

type Vector2D(dx:double,dy:double) =
    member v.DX = dx
    member v.DY = dy
    member v.Length = sqrt(dx*dx+dy*dy)
    member v.Scale(k) = Vector2D(dx*k,dy*k)

```

Inputs to  
object  
construction

Exported  
properties

Exported  
method

## F# - Objects + Functional

```

type Vector2D(dx:double,dy:double) =

```

```

    let norm2 = dx*dx+dy*dy

```

Internal (pre-  
computed) values  
and functions

```

    member v.DX = dx

```

```

    member v.DY = dy

```

```

    member v.Length = sqrt(norm2)

```

```

    member v.Norm2 = norm2

```

## F# - Objects + Functional

```
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)
```

Immutable  
inputs

Internal  
tables

Publish  
access

## F# - Objects + Functional

```
type Vector2D(dx:double,dy:double) =
```

```
    let mutable currDX = dx
```

```
    let mutable currDY = dy
```

```
    member v.DX = currDX
```

```
    member v.DY = currDY
```

```
    member v.Move(x,y) =  
        currDX <- currDX+x  
        currDY <- currDY+y
```

Internal state

Publish  
internal state

Mutate internal  
state

## F# - Language Oriented

```

type PropLogic =
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True

let rec Eval(prop) =
    match prop with
    | And(a,b) -> Eval(a) && Eval(b)
    | Not(a) -> not (Eval(a))
    | True -> true

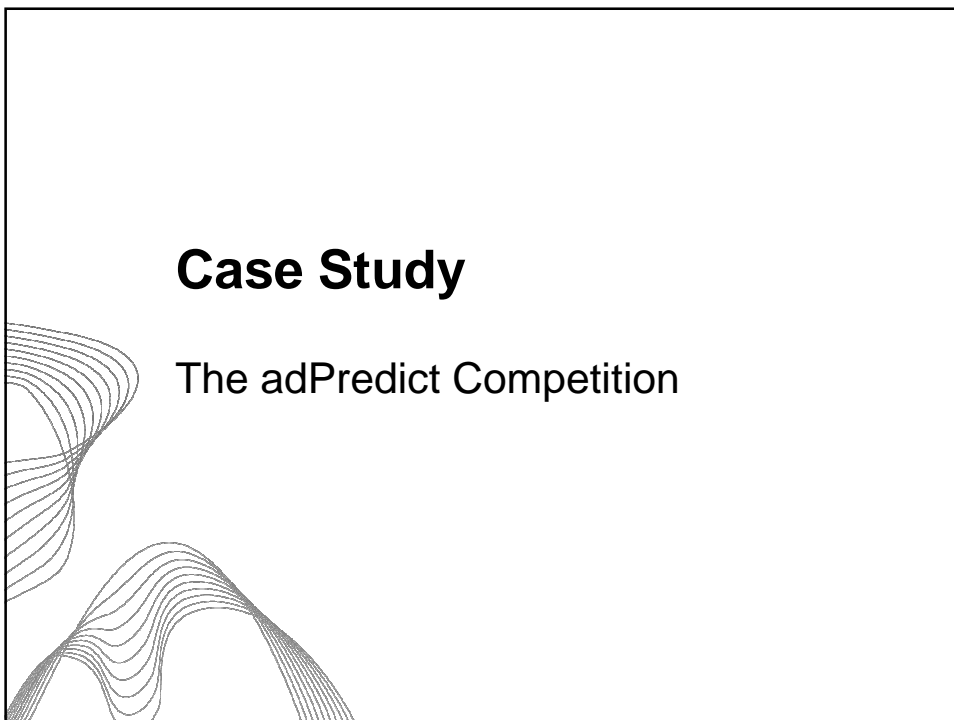
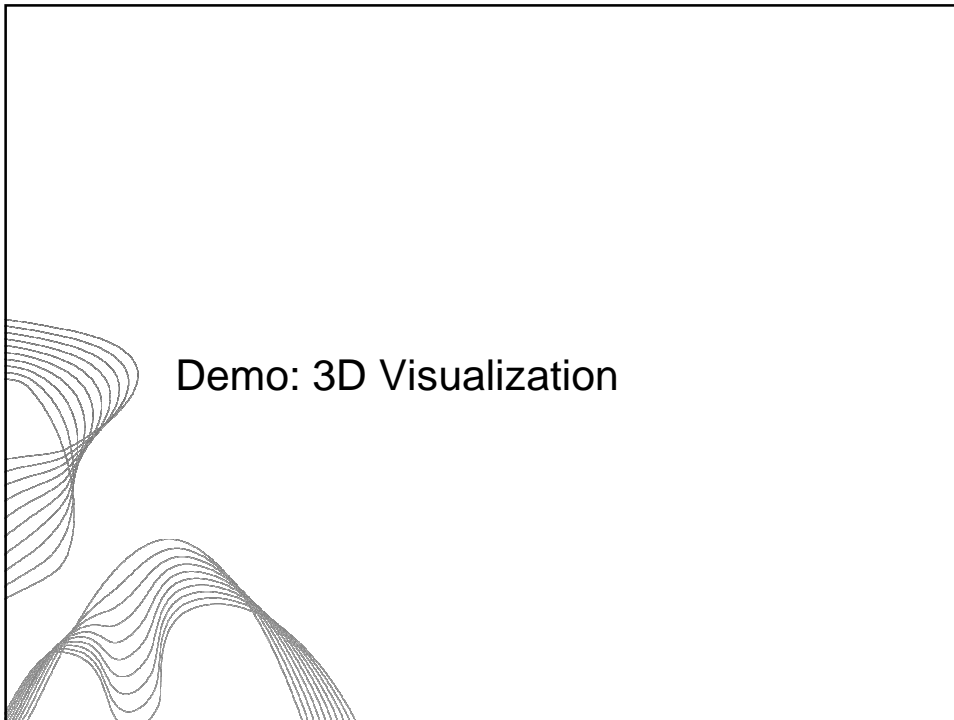
```

Embedded  
Language

Crisp  
Semantics

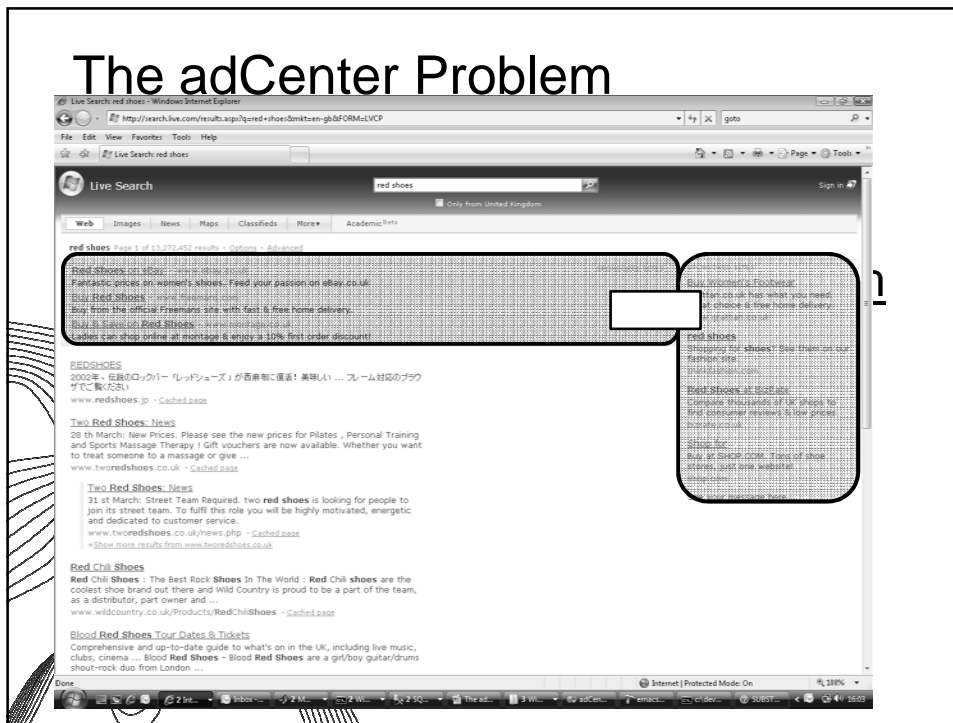
## F# - What's it For?

- Language: *Math like, succinct, functional*
  - Users have a mathematical background
  - Shun infrastructure and "boilerplate"
- Quality: *Performant, scalable, reliable, supported*
  - Live and mission critical apps
  - Huge data loads
- Tool: *Explorative, interactive, connected*
  - On the cutting edge, working in uncharted territory
  - Experimentation and tuning are important
  - Acquire and analyze multiple data sources
- .NET Libraries: *HPC, Concurrency*
  - Heavy on computation, analysis, prediction, parallelism





## The adCenter Problem



## The Scale of Things

- **Weeks of data in training:**  
7,000,000,000 impressions, 6TB data
- **2 weeks of CPU time during training:**  

$$2 \text{ wks} \times 7 \text{ days} \times 86,400 \text{ sec/day} =$$

$$1,209,600 \text{ seconds}$$
- **Learning algorithm speed requirement:**
  - 5,787 impression updates / sec
  - 172.8  $\mu$ s per impression update

## F# and adCenter

- 4 week project, 4 machine learning experts
- 100million probabilistic variables
- Processes 6TB of training data
- Real time processing

## AdPredict: What We O

- Quick Coding
- Agile Coding
- Scripting
- Performance
- Memory-Faithful
- Succinct
- Symbolic
- .NET Integration

F#'s powerful type inference means less typing, more thinking

Type-inferred code is easily refactored

"Hands-on" exploration.

Immediate scaling to massive data sets

mega-data structures, 16GB machines

Live in the **domain**, not the language

Schema compilation and "Schedules"

Especially Excel, SQL Server

## F# - Concurrent/Reactive/Parallel

- **Concurrent**: *Multiple threads* of execution
- **Parallel**: These execute *simultaneously*
- **Asynchronous**: Computations that complete "*later*"
- **Reactive**: *Waiting and responding* is normal

## Why is it so hard?

- To get 50 web pages in parallel?
- To get from thread to thread?
- To create a worker thread that reads messages?
- To handle failure on worker threads?

## Why isn't it this easy?

```
let ProcessImages() =  
    Async.Run  
        (Async.Parallel  
            [ for i in 1 .. numImages -> ProcessImage(i) ])
```

## Why isn't it this easy?

```
let task =  
    async { ...  
        do! SwitchToNewThread()  
        ...  
        do! SwitchToThreadPool()  
        ...  
        do! SwitchToGuiThread()  
        .... }  
}
```

## Some Foundation Technologies

- .NET/Win32 Threads
- System.Threading
- .NET Thread Pool
- .NET BackgroundWorker and SynchronizationContexts
- Parallel Extensions for .NET

## F# - Concurrent/Reactive/Parallel

- **Concurrent**: *Multiple threads of execution*
- **Parallel**: *These execute *simultaneously**
- **Asynchronous**: *Computations that complete "*later*"*
- **Reactive**: *Waiting and responding is normal*

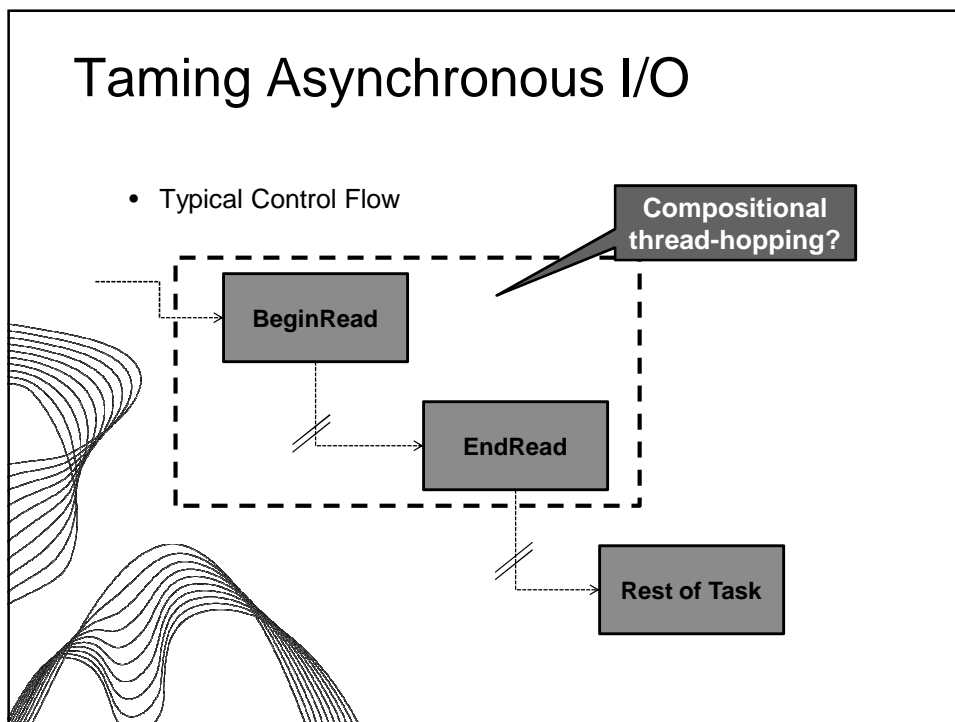
## Taming Asynchronous I/O

Target: make it easy to use Begin/End operations

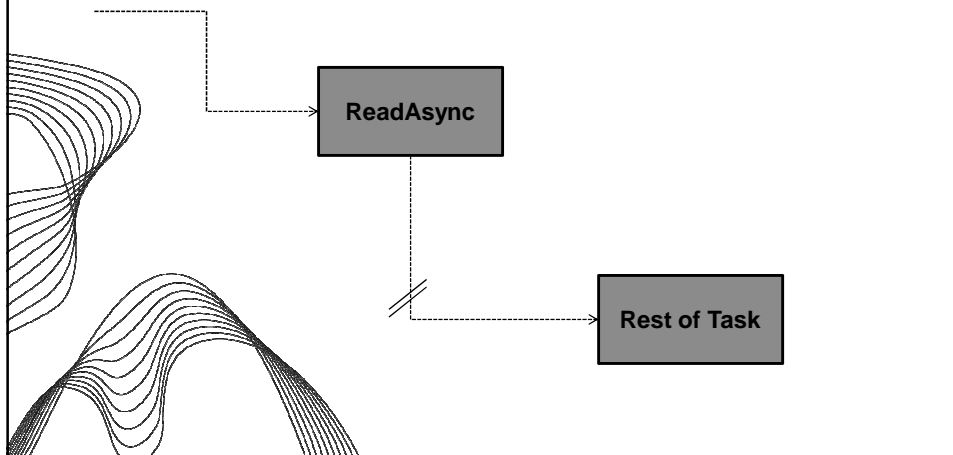
```

inStream. (pixels,0,numPixels,
BeginRead
BeginWrite
CanRead
CanSeek
CanTimeout
CanWrite
Close
CreateObjRef
CreateWaitHandle
Dispose
CCallback(fun iar -> conti
e=null) |> ignore
hed
// Wait un
ContinueEv
Stream.BeginRead : ...
Stream.EndRead : IAsyncResult * ...
    
```

## Taming Asynchronous I/O



## Taming Asynchronous I/O



## Simple Examples

```
Async.Parallel [ async { -> 2*2 + 3*6 };
                 async { -> 3 + 5 - 1 } ]
```

Compute 22  
and 7 in  
parallel

```
Async.Parallel [WebRequest.Async "http://www.live.com",
                 WebRequest.Async "http://www.yahoo.com",
                 WebRequest.Async "http://www.google.com" ]
```

Get these  
three web  
pages and wait  
until all have  
come back

```
let parArrMap f (arr: _[]) =
    Async.Run (Async.Parallel [| for x in arr -> async { -> f x } |])
```

Naive Parallel Array  
Map

# Taming Asynchronous I/O

```

using System;
using System.IO;
using System.Threading;

public class BulkImageProcAsync
{
    public const String ImageBaseName = "Image";
    public const int numImages = 200;
    public const int numPixels = 512;

    // ProcessImage has a simple O(N^2)
    // of times you repeat that loop
    // bound on more IO-bound.
    public static int processImage(
        ImageStateObj state,
        ImageStateObj asyncRes)
    {
        // Using asynchronous I/O here appears not to be
        // It ends up swamping the threadpool, because the
        // threads are blocked on I/O requests that were
        // the threadpool.
        FileStream fs = new FileStream(ImageBaseName + state.imageNum,
            FileMode.Create, FileAccess.Write, FileShare.None,
            4096, false);
        fs.Write(state.pixels, 0, numPixels);
        fs.Close();
    }

    public static void ReadInImageCallback(IAsyncResult asyncRes)
    {
        ImageStateObj state = (ImageStateObj)asyncRes.AsyncState;
        Stream stream = state.fs;
        int bytesRead = stream.EndRead(asyncRes);
        if (bytesRead != numPixels)
            throw new Exception(String.Format(
                "In ReadInImageCallback, got the wrong number of bytes from the image: {0}.", bytesRead));
        ProcessImage(state.pixels, state.imageNum);
        stream.Close();
    }

    public static void ProcessImagesInBulk()
    {
        Console.WriteLine("Processing images...");
        long t0 = Environment.TickCount;
        NumImagesToFinish = numImages;
        AsyncCallback readImageCallback = new AsyncCallback(ReadInImageCallback);
        for (int i = 0; i < numImages; i++)
        {
            ImageStateObj state = new ImageStateObj();
            state.pixels = new byte[numPixels];
            state.imageNum = i;
            // Very large items are read only once, so you can make the
            // buffer on the FileStream very small to save memory.
            FileStream fs = new FileStream(ImageBaseName + i + ".tmp",
                FileMode.Open, FileAccess.Read, FileShare.Read, 1, true);
            state.fs = fs;
            fs.BeginRead(state.pixels, 0, numPixels, readImageCallback,
                state);
        }

        // Determine whether all images are done being processed.
        // This blocks until all are finished.
        bool mustBlock = true;
        lock (NumImagesToFinish)
        {
            if (NumImagesToFinish > 0)
                mustBlock = true;
        }
        if (mustBlock)
        {
            Console.WriteLine("All work done. Blocking until they complete. numLeft: {0}",
                NumImagesToFinish);
            Monitor.Enter(waitObject);
            Monitor.Wait(waitObject);
            Monitor.Exit(waitObject);
        }
        long t1 = Environment.TickCount;
        Console.WriteLine("Total time processing images: {0}ms",
            (t1 - t0));
    }
}
    
```

**Processing 200 images in parallel**

## 8 Ways to Learn

- **FSI.exe**
- **<http://cs.hubfs.net>**
- **Samples Included**
- **Codeplex Fsharp Samples**
- **Go to definition**
- **Books**
- **Lutz' Reflector**
- **ML**



## Books about F#



Visit <http://research.microsoft.com/fsharp>  
<http://blogs.msdn.com/dsyme>

## Getting F#

- “Spring Refresh” just released (1.9.4)
  - Focus on Simplicity, Regularity, Correctness
- CTP: Late Summer
- And then onwards towards “Visual Studio Next”

