Winner

Alan Turing v Alonzo Church

Cage Match to the Death!
The Von Neumann bottleneck
Moore’s Law to the Rescue?
Agenda

- Intro to Functional Programming
- Intro to F#
- Functional Programming via F#
  - Higher order functions
  - Language Oriented Programming
  - Asynchronous workflows
What is **object-oriented** programming?

**Object-oriented** programming is a style of programming that enables you:
- Reuse code (via classes)
- Eliminate bugs (via encapsulating, data hiding)

An **object-oriented** language is one which supports object-oriented programming natively.

**C#** is a popular **object-oriented** language for .NET
What is **functional** programming?

**Functional** programming is a style of programming that enables you:
- Reuse code (via function composition)
- Eliminate bugs (via immutability)

A **functional** language is one which functional programming natively.

**F#** is a popular **functional** language for .NET.
Functional Programming

- Emphasis is on what is to be computed not how it happens
- Data is immutable
- Functions are data too
What is to be computed, not how.

Pattern Matching

```haskell
// Fibonacci numbers
// 1, 1, 2, 3, 5, 8, 13, ...
let rec fibonacci =
  function
  | 0 | 1 -> 1
  | 2 -> 2
  | 3 -> 3
  | 4 -> 5
  | x -> fibonacci (x - 1) + fibonacci (x - 2)
```

Discriminated Unions

```haskell
type Suit = Heart | Diamond | Club | Spade
type Card =
  | Ace of Suit
  | King of Suit
  | Queen of Suit
  | Jack of Suit
  | ValueCard of int * Suit

let deck = [
  for suit in [Heart; Diamond; Club; Spade] do
    for value in 2 .. 10 do
      yield ValueCard(value, suit)
  yield Jack(suit); yield Queen(suit)
  yield King(suit); yield Ace(suit)
]
```

Pattern Matching

```haskell
// Size of the 'My Pictures' folder in MB
let sizeOfMyPicturesFolder =
  filesUnderFolder (GetFolderPath(SpecialFolder.MyPictures))
  |> Seq.map GetFileInfo
  |> Seq.map GetFileSize
  |> Seq.fold (+) 0L
  |> ConvertBytesToMB
```

Haskell is really good at being expressive...

```
qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

List Comprehensions
Data is immutable

\[ x = x + 1; \]
Data is immutable (continued)

- Why should a function in C never return a pointer?

- Why should you make a copy of an internal array before returning it from your class?

- Why is multi-threading so damn hard?
Functions are data too

Let \(\text{squares} = \text{List.map (fun i -> i * i) [1..10]}\)
\(\Rightarrow \text{squares} = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]\)

Let \(\text{IsEven} x = (x \mod 2 = 0)\)
\(\text{let evens = Array.filter IsEven [| 1 .. 10 |]}\)

Let recursive \(\text{ForLoop startidx endidx f} = \)
\(f()\)
\(\text{if startidx = endidx}\)
\(\text{then} ()\)
\(\text{else ForLoop (startidx + 1) endidx f}\)

// Append text to a file
let AppendFile (fileName : string) (text : string) =
  let file = new StreamWriter(fileName, true) in
  file.WriteLine(text) in
  file.Close()

// Append data to Log.txt
AppendFile @"D:\Log.txt" "Processing Event X...";

// Only provide the first parameter, Log.txt
let CurriedAppendFile = AppendFile @"D:\Log.txt"

// Append the text to Log.txt
CurriedAppendFile "Processing Event Y..."
Agenda

- Intro to Functional Programming
- **Intro to F#**
- What functional programming offers
  - Higher order functions
  - Language Oriented Programming
  - Asynchronous workflows
Introducing F#

F# is a .NET programming language

F# is
- Functional
- Imperative
- Object Oriented
Imperative

- Side effects

- Ability to declare and mutate variables

```fsharp
let mutable x = ""
x <- "Hello, World"
printfn "%s" x
```

- Control flow (while, for, if, exceptions, etc.)

```fsharp
// No statements in F#, if-expressions
let thingsToDoToday =
    if DateTime.Now.DayOfWeek = DayOfWeek.Wednesday then
        [ givePresentation() ]
    else
        [ workOnTalk(); workOnSlides() ]
```
Object Oriented (and .NET)

- Classes and Interfaces
- Polymorphism and Inheritance
- Delegates and Events
- Structs and Enums
First-class Citizen of .NET

- **Tools**
  - Visual Studio
  - FXCop
  - SQL Server

- **Libraries**
  - Managed DirectX
  - Visual Studio Tools for Office
  - WinForms, WCF, WPF
Agenda

- Intro to Functional Programming
- Intro to F#
- What functional programming offers
  - Functions, Records, and Discriminated Unions
  - Language Oriented Programming
  - Asynchronous workflows
Onto Visual Studio!
Language Oriented Programming
The language is in the data

Concrete Representations

Abstract Representations

Integrated Representations

LOP Taxonomy

XML, CSV, Text, Strings, JSON

Parse Trees

Almost-Implicit Parallelism

Queries

Exception Handling

Workflows

The language is in the code
LOP Techniques

Concrete Representations

Abstract Representations

Integrated Representations

XML Libraries
RegExp Libraries
Lex/Yacc
...

Discriminated Unions

Pattern Matching

F# Computation Expressions

Expression Trees
Onto Visual Studio!
Async Workflows

AKA Computation Expressions
Definitions

- **Parallel** — Start doing several things at once, wait until all of them are done before continuing.

- **Asynchronous** — Start doing something in the background, it will notify you when it’s finished.

- **Reactive** — Something just sits there and when you need something from it, it will respond.
Taming Asynchronous I/O

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

public class BulkImageProcAsync
{
    public const String ImageBaseName = "Image%d"
    public const int numImages = 200
    public const int numPixels = 512 * 512

    // ProcessImage has a simple O(N) loop, and you can vary the number
    // Threads must decrement NumImagesToFinish
    // Their access to it through a
    // buffer on the FileStream very small to save memory.
    public static Object[] ImageStateObject
    
    public static void ReadInImageCallback(IAsyncResult asynResult)
    {
        ImageStateObject state = (ImageStateObject)asynResult.AsyncState;
        Stream stream = state.fs;
        int bytesRead = stream.EndRead(asynResult);
        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();

        // Now write out the image.
        // Using asynchronous I/O here appears not to be best practice.
        // It ends up swamping the threadpool, because the
        // threads are blocked on I/O requests that were just queued to
        // the threadpool.
        FileStream fs = new FileStream(ImageBaseName + state.imageNum + ".done", FileMode.Create, FileAccess.Write, FileShare.None);
        fs.Write(state.pixels, 0, numPixels);
        fs.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++)
        {
            ImageStateObject state = new ImageStateObject();
            state.pixels = new byte[numPixels];
            state.imageNum = i;
            state.fs = 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.
        // If not, block until all are finished.
        bool mustBlock = true;
        lock (NumImagesMutex)
        {
            if (NumImagesToFinish > 0)
                mustBlock = true;
        }
        if (mustBlock)
        {
            Console.WriteLine("All worker threads are queued. 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));
    }
}
Taming Asynchronous I/O

Equivalent F# code (same perf)

```fsharp
let ProcessImageAsync(i) =
    async {
        use inStream = File.OpenRead(sprintf "source%d.jpg" i)
        let! pixels = inStream.ReadAsync(numPixels)
        let pixels' = TransformImage(pixels, i)
        use outStream = File.OpenWrite(sprintf "result%d.jpg" i)
        do! outStream.WriteAsync(pixels')
        do Console.WriteLine "done!"
    }

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

This object coordinates

```
"!" = "asynchronous"
```

Open the file, synchronously

```
OpenRead(sprintf "source%d.jpg" i)
```

Read from the file, asynchronously

```
ReadAsync(numPixels)
```

Generate the tasks and queue them in parallel

```
Async.Parallel
```

Write the result, asynchronously

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

// Now write out the image.
// Using asynchronous I/O here appears not to be best practice.
// It ends up swamping the threadpool, because the threads are blocked on I/O requests that were just
// threads are blocked on I/O requests that were just
// their access to it through a global NumImagesToFinish

state.fs = fs;
fs.BeginRead(state.pixels, 0, numPixels, readImageCallback, state);

// Determine whether all images are done being processed.
// Record that an image is finished now.

public static void ProcessImagesAsyncWorkflow()
{
    Async.Run (Async.Parallel
        [ for i in 1 .. numImages -> ProcessImageAsync i ],
        null)
}

public static async void ProcessImageAsync()
{
    var asyncResult = await Async.Parallel
        [ for i = 1 .. numImagesToFinish
            async do
                let pixels = await transforms.TransformImage (inStream.ReadAsync (numPixels))
                let pixels' = TransformImage (pixels, i)
                let outStream = File.OpenWrite (sprintf Image%d.done % i)
                outStream.WriteAsync (pixels')
                Monitor.Pulse (NumImagesMutex) ];

    Monitor.Wait (WaitObject);
    // All images are done now.
    fs.Close();
}

public static void TransformImage (byte[] pixels, int i)
{
    // Transform the image, however you like.
    // Do it here.
    // It ends up swamping the threadpool.
    // Using asynchronous I/O here appears not to be best practice.
    // It ends up swamping the threadpool, because the threads are blocked on I/O requests that were just
    // threads are blocked on I/O requests that were just
    // their access to it through a global NumImagesToFinish
    // Record that an image is finished now.
}

public class BulkImageProcAsync
{
    public const String ImageBaseName = "tmpImage");
    public const int numImages = 200;
    public const int numPixels = 512 * 512;
    public const int processImageRepeats = 20;
    public static Object[] WaitObject = new Object[0];
    public static Object[] NumImagesMutex = new Object[0];
    // WaitObject is signalled when all image processing is done.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    // NumImagesToFinish
    // используется для синхронизации между потоками.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    public void ProcessImageAsync()
    {
        Async.Run (Async.Parallel
            [ for i = 1 .. numImagesToFinish
                async do
                    let pixels = await transforms.TransformImage (inStream.ReadAsync (numPixels))
                    let pixels' = TransformImage (pixels, i)
                    let outStream = File.OpenWrite (sprintf Image%d.done % i)
                    outStream.WriteAsync (pixels')
                    Monitor.Pulse (NumImagesMutex) ];
        Monitor.Wait (WaitObject);
        // All images are done now.
        fs.Close();
    }
}

public static void TransformImage (byte[] pixels, int i)
{
    // Transform the image, however you like.
    // Do it here.
    // It ends up swamping the threadpool.
    // Using asynchronous I/O here appears not to be best practice.
    // It ends up swamping the threadpool, because the threads are blocked on I/O requests that were just
    // threads are blocked on I/O requests that were just
    // their access to it through a global NumImagesToFinish
    // Record that an image is finished now.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    // NumImagesToFinish
    // используется для синхронизации между потоками.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    public void ProcessImageAsync()
    {
        Async.Run (Async.Parallel
            [ for i = 1 .. numImagesToFinish
                async do
                    let pixels = await transforms.TransformImage (inStream.ReadAsync (numPixels))
                    let pixels' = TransformImage (pixels, i)
                    let outStream = File.OpenWrite (sprintf Image%d.done % i)
                    outStream.WriteAsync (pixels')
                    Monitor.Pulse (NumImagesMutex) ];
        Monitor.Wait (WaitObject);
        // All images are done now.
        fs.Close();
    }
}

public static void TransformImage (byte[] pixels, int i)
{
    // Transform the image, however you like.
    // Do it here.
    // It ends up swamping the threadpool.
    // Using asynchronous I/O here appears not to be best practice.
    // It ends up swamping the threadpool, because the threads are blocked on I/O requests that were just
    // threads are blocked on I/O requests that were just
    // their access to it through a global NumImagesToFinish
    // Record that an image is finished now.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it through a mutex.
    public void ProcessImageAsync()
    {
        Async.Run (Async.Parallel
            [ for i = 1 .. numImagesToFinish
                async do
                    let pixels = await transforms.TransformImage (inStream.ReadAsync (numPixels))
                    let pixels' = TransformImage (pixels, i)
                    let outStream = File.OpenWrite (sprintf Image%d.done % i)
                    outStream.WriteAsync (pixels')
                    Monitor.Pulse (NumImagesMutex) ];
        Monitor.Wait (WaitObject);
        // All images are done now.
        fs.Close();
    }
}
How does it work?

- Uses Computational LOP to make writing continuation-passing programs simpler and compositional

- Similar to techniques used in Haskell

- A wrapper over the .NET Thread Pool and .NET synchronization primitives
F# “Workflow” Syntax

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

You're actually writing this (approximately):

```fsharp
async.Delay(fun () ->
    async.Bind(readAsync "cat.jpg", (fun image ->
        async.Bind(async.Return(f image), (fun image2 ->
            async.Bind(writeAsync "dog.jpg", (fun () ->
                async.Bind(async.Return(printfn "done!"), (fun () ->
                    async.Return(())))))))))))
```
Onto Visual Studio!
F# Resources

- **Getting F#**
  - Download from [http://msdn.com/fsharp](http://msdn.com/fsharp)
  - Installs as an Add-In for Visual Studio 2008

- **Books**
  - *Expert F#*
    - Don Syme, Adam Granicz, and Antonio Cisternino
  - *Foundations of F#*
    - Robert Pickering
  - *F# for Scientists*
    - John Harrop

- **Websites**
  - [http://cs.hubfs.net/](http://cs.hubfs.net/)
Questions

http://cs.hubfs.net
http://msdn.com/fsharp
http://blogs.msdn.com/chrsmith