Smart Software with F#

Joel Pobar

http://callvirt.net/blog

Agenda

- **Why Functional Programming?**
- F# Language Walkthrough
- Smart Software Ideas
  - Search
  - Fuzzy Matching
  - Classification
  - Recommendations

F# is...

...a **functional, object-oriented, imperative and explorative** programming language for .NET

what is Functional Programming?

What is FP?

- Wikipedia: “A programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data”

- -> Emphasizes functions
- -> Emphasizes shapes of data, rather than impl.
- -> Modeled on lambda calculus
- -> Reduced emphasis on imperative
- -> Safely raises level of abstraction

Movation

- Simplicity in life is good: cheaper, faster, better.
  - We typically achieve ***simplicity*** by:
    - By raising the level of ***abstraction***
    - Increasing ***modularity***
    - Increasing ***expressiveness*** (signal to noise)
- Composition and modularity == reuse
- Environment is changing: safety, concurrency, non-deterministic, non-sequential

Composition, Modularity

Why Functional Programming Matters

John Hughes, Institutionen för Datavetenskap,
Chalmers Tekniska Högskola,
41296 Göteborg,
SWEDEN. rjmh@cs.chalmers.se

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs. Conventional languages place conceptual limits on the way problems can be modularised. Functional languages push those limits back. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity. As examples, we manipu-

Composition, Modularity

- **Reg Braithwaite:** *“Let’s ask a question about Monopoly (and Enterprise Software). Where do the rules live? In a noun-oriented design, the rules are smooshed and smeared across the design, because every single object is responsible for knowing everything about everything it can ‘do’. All the verbs are glued to the nouns as methods.”*

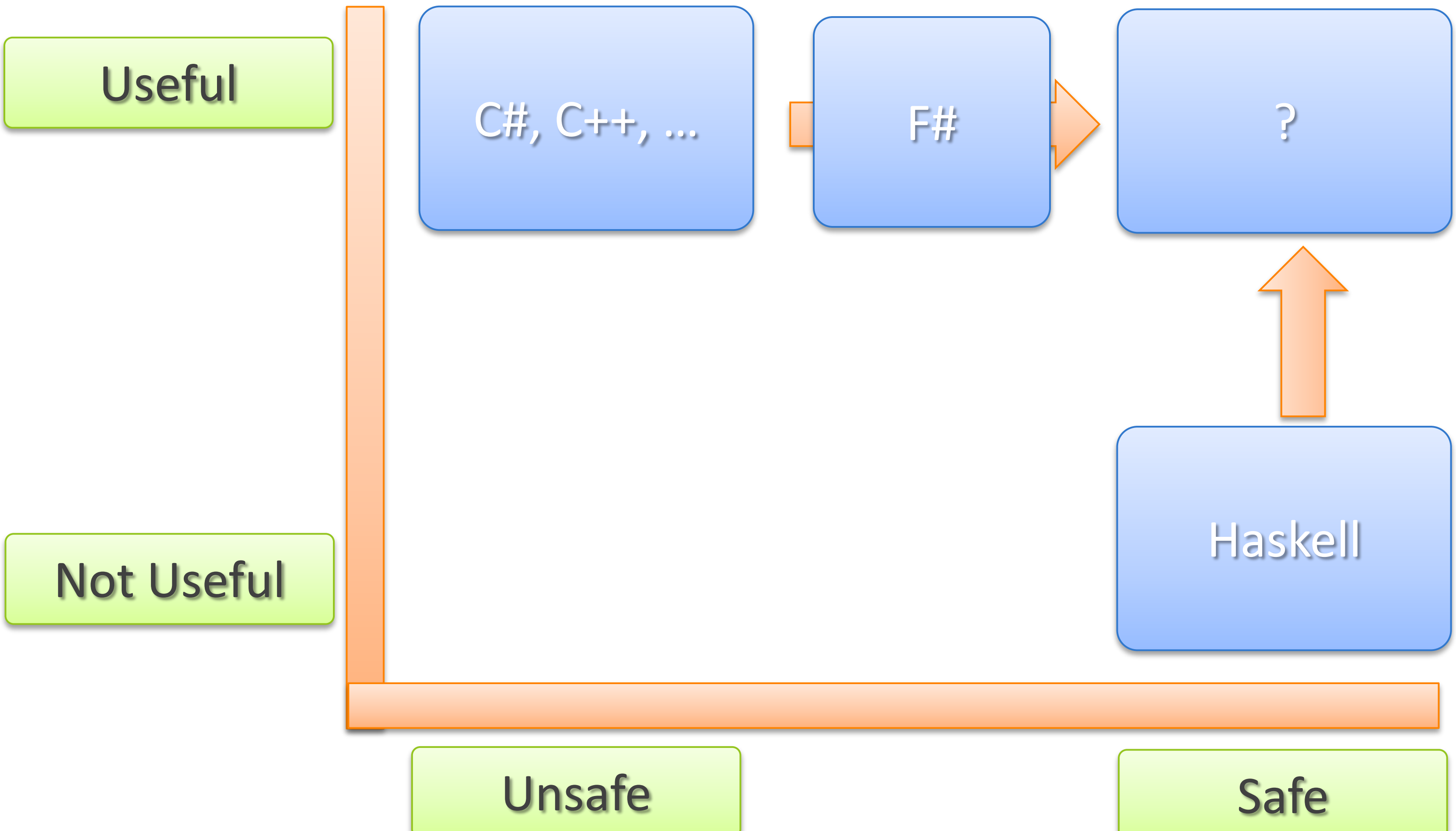
Composition, Modularity

- *“The great insight is that better programs separate concerns. They are factored more purely, and the factors are naturally along the lines of responsibility (rather than in Jenga piles of abstract virtual base mixin module class proto_ extends private implements). Languages that facilitate better separation of concerns are more powerful in practice than those that don’t.”*

Changing Environment

- Multi-core
  - Moore’s law still strong, the old “clock frequency” corollary isn’t...
- Distributed computing
  - The cloud is the next big thing
  - Millisecond computations
  - Async
  - Local vs. Remote computation (mobile devices)
- Massive data
- DSL’s: raising abstraction for consumers of software
- Risk!
  - Big budgets, small timeframes, and reliability as first class!

Safe and Useful



Agenda

- Why Functional Programming?
- **F# Language Walkthrough**
- Smart Software Ideas
  - Recommendations
  - Fuzzy Matching
  - Search
  - Classification

F# Overview

F# is a **.NET** language

F# is a **multi-paradigm** language

F# is a **statically-typed** language

F# is a **feature-rich** language

F# Overview

F# is a **.NET** language

- Runs on any CLI implementation (including Mono!)
- Consumes any .NET library
- Interoperates with any .NET language

F# is a **multi-paradigm** language

F# is a **statically-typed** language

F# is a **feature-rich** language

F# Overview

F# is a **.NET** language

F# is a **multi-paradigm** language

- Embraces functional, imperative and OO paradigms
- Encourages a functional programming style

F# is a **statically-typed** language

F# is a **feature-rich** language

F# Overview

F# is a **.NET** language

F# is a **multi-paradigm** language

F# is a **statically-typed** language

- **RICH** type inference
- Expect to see very few type annotations
- **NOT** a dynamically-typed language

F# is a **feature-rich** language

F# Overview

F# is a **.NET** language

F# is a **multi-paradigm** language

F# is a **statically-typed** language

F# is a **feature-rich** language

- Broad, rich type system
- Control computation semantics
- Hack on the compiler AST
- Read-Eval-Print-Loop (REPL) & scripting support
- ML style functional programming library (FSharp.Core.dll)

F# Syntax

let *binding values to names*

```
let hello = “Hello World”
```

Type inferred

```
let numbers = [1 .. 10]
let odds = [1; 3; 5; 7; 9]
let evens = [0 .. 2 .. 10]
let squares = [for x in numbers -> x * x]
```

```
let a = 10
let a = 20  // error
```

Immutable by default

Functions

fun *functions as values*

Type inferred

```
let square x = x * x
let add x y = x + y
```

```
let squares = List.map (fun x -> x * x) [1..10]
```

```
let squares = List.map square [1..10]
```

Functions

Space is important – 4 spaces gives you scope

```
let appendFile (fileName: string) (text: string) =
    use file = new StreamWriter(fileName, true)
    file.WriteLine(text)
    file.Close()

val appendFile : string -> string -> unit
```

Function signature says: "The first parameter is a string, and the result is a function which takes a string and returns unit". Enables a powerful feature called "currying"

Tuple

`( , )` *fundamental type*

```
> let dinner = (“eggs”, “ham”)

val dinner: string * string = (“eggs”, “ham”)

> let entree, main = dinner
```

```
> let zeros = (0, 0L, 0I, 0.0)

val zeros: int * int64 * bigint * float = ...
```

Lists

`[;]` *fundamental type*

```
let vowels = [‘a’; ‘e’; ‘i’; ‘o’; ‘u’]
let emptyList = []

let sometimes = ‘y’ :: vowels        // cons
let others = [‘z’; ‘x’;] @ vowels   // append
```

Lists

`List.length`	`‘a list -> int`
`List.head`	`‘a list -> ‘a`
`List.tail`	`‘a list -> ‘a`
`List.exists`	`(‘a -> bool) -> ‘a list -> bool`
`List.rev`	`‘a list -> ‘a list`
`List.tryFind`	`(‘a -> bool) -> ‘a list -> ‘a option`
`List.filter`	`(‘a -> bool) -> ‘a list -> ‘a list`
`List.partition`	`(‘a -> bool) -> ‘a list -> (‘a list * ‘a list)`
`...`	

Lists

Aggregate operations

```
List.iter
      (‘a -> unit) -> ‘a list -> unit
List.map
      (‘a -> ‘b) -> ‘a list -> ‘b list
List.reduce
      (‘a -> ‘a -> ‘a) -> ‘a list -> ‘a
List.fold
      (‘a -> ‘b -> ‘a) -> ‘a -> ‘b list -> ‘a
...
```

Sequences

`seq {}` *lazy evaluation*

```
let seqOfNumbers = seq { 1 .. 10000000 }
let alpha = seq { for c in ‘A’ .. ‘z’ -> c }

let rec allFilesUnder basePath =
      seq {
            yield! Directory.GetFiles(basePath)
            for subDir in
Directory.GetDirectories(basePath) do
                  yield! allFilesUnder subDir
          }
```

F# Syntax

|> *bringing order to chaos*

```
let sumOfSquares =
  List.sum (List.map square [1..10])
```

```
let (|>) x f = f x
```

```
let sumOfSquares = [1..10]
                   |> List.map square
                   |> List.sum
```

Types

`type` *discriminated unions*

```
type Suit = | Spade | Heart | Club | Diamond
```

```
type Rank = | Ace | King | Queen | Jack
            | Value of int
```

```
type Card = Card of Suit * Rank
```

Types

`type` *records*

```
type Person =
      { First: string; Last: string; Age: int}

let b = {First = “Bill”; Last = “Gates”; Age =
54}

printfn “%s is %d years old” b.First b.Age
```

Types

type *classes (.NET ref types)*

```
type Point =
     val m_x : float
     val m_y : float

     new (x, y) = { m_x = x; m_y = y }
     new () = { m_x = 0.0; m_y = 0.0 }

     member this.Length =
          let sqr x = x * x
          sqrt <| sqr this.m_x + sqr this.m_y
```

Types

`interface` *.NET interfaces*

```
type IWriteScreen =
    abstract member Print : string -> unit

type SomeClass =
    interface IWriteScreen with
        member this.Print (str: string) =
Console.WriteLine(str)
```

F# Syntax

`match` *pattern matching*

```
let cardValue (Card(r,s)) =
  match r with
  | Ace                  -> 11
  | King | Queen | Jack  -> 10
  | Value(x)             -> x
```

```
let oddOrEven x =
  match x with
  | x when x % 2 = 0 -> "even"
  | _                -> "odd"
```

Demo: Quick Lap around F#

Agenda

- Why Functional Programming?
- F# Language Walkthrough
- **Smart Software Ideas**
  - Recommendations
  - Fuzzy Matching
  - Search
  - Classification

Recommendation Engine

- Netflix Prize - $1 million USD
  - Must beat Netflix prediction algorithm by 10%
  - 480k users
  - 100 million ratings
  - 18,000 movies
- Great example of deriving value out of large datasets
- Earns Netflix loads and loads of $$$!

- Unfortunately no longer running:
  - Instead we'll be using the MovieLens dataset

MovieLens Data Format

MovieId	CustomerId	Rating
Clerks	444444	5.0
Clerks	2093393	4.5
Clerks	999	5.0
Clerks	8668478	1.0
Dogma	2432114	3.0
Dogma	444444	5.0
Dogma	999	5.0
...	...	...

Nearest Neighbour

MovieId	CustomerId	Rating
Clerks	444444	5.0
Clerks	2093393	4.0
Clerks	999	5.0
Clerks	8668478	1.0
Dogma	2432114	3.0
Dogma	444444	5.0
Dogma	999	5.0
...	...	...

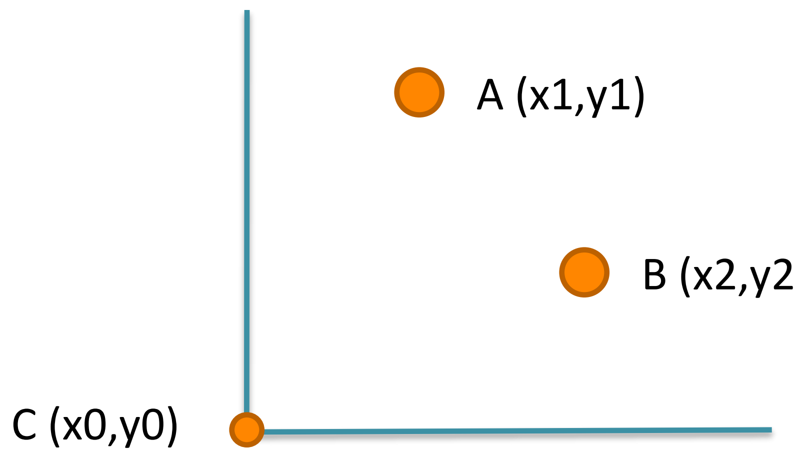
Recommendation Engine

- Find the best movies my neighbours agree on:

CustomerId	302	4418	3	56	732
444444	5	4	5		2
999	5		5	1	
111211		3	5		3
66666	5		5		
1212121	5		4		
5656565					1
454545	5		5		

Demo: Recommendation Engine

Nearest Neighbour: Vector Math



- If we want to calculate the distance between A and B, we call on Euclidean Distance
- We can represent the points in the same way using Vectors: Magnitude and Direction.
- Having this Vector representation, allows us to work in ‘n’ dimensions, yet still achieve
- Euclidean Distance/Angle calculations.

Agenda

- Why Functional Programming?
- F# Language Walkthrough
- **Smart Software Ideas**
  - Recommendations
  - Fuzzy Matching
  - Search
  - Classification

Fuzzy Matching

- String similarity algorithms:
  - SoundEx; Metaphone
  - Jaro Winkler Distance; Cosine similarity; Sellers; Euclidean distance; …
  - We'll look at Levenshtein Distance algorithm

- Defined as: *The minimum edit operations which transforms string1 into string2*

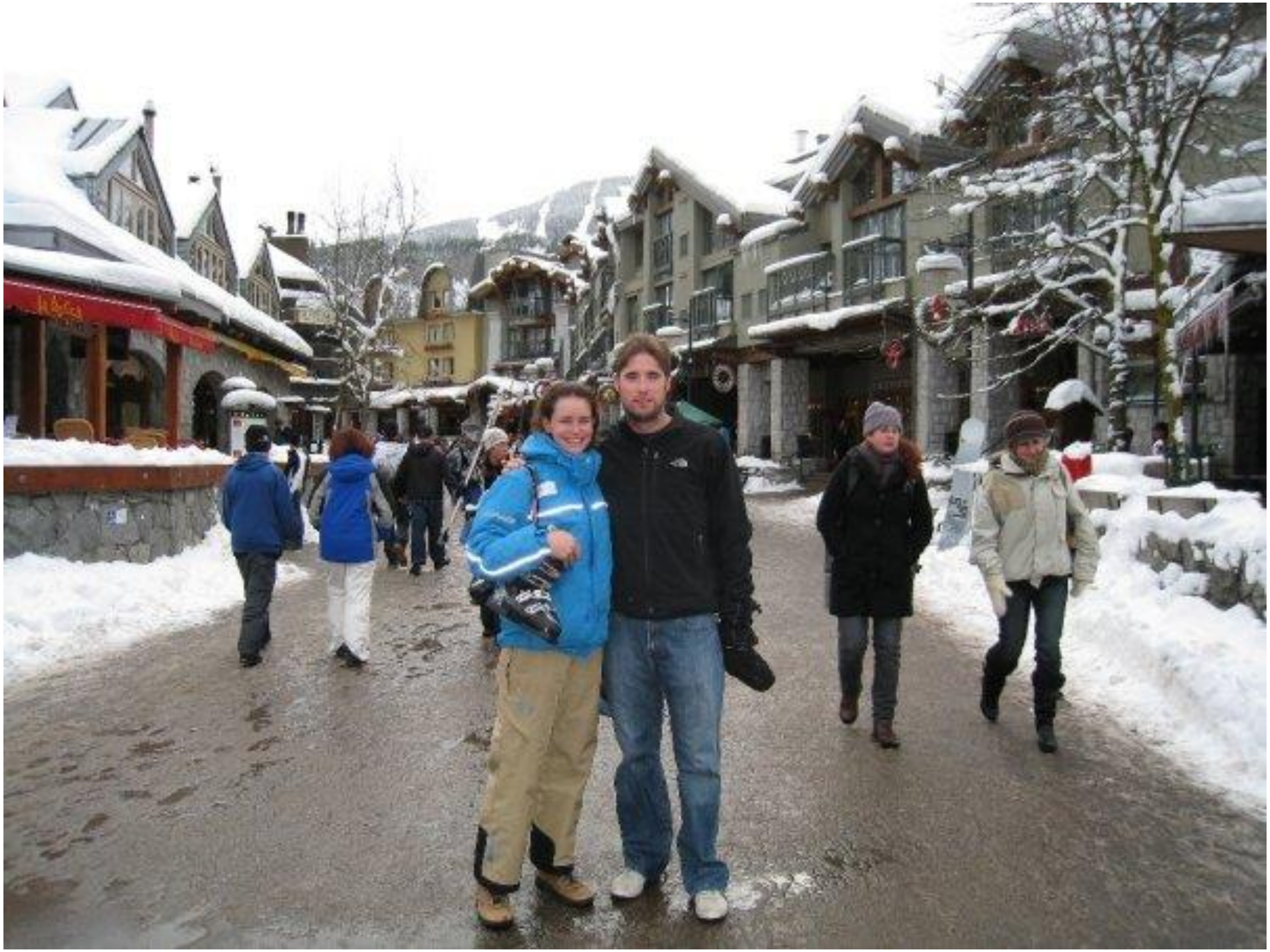
Fuzzy Matching

- Edit costs:
  - In-place copy – cost 0
  - Delete a character in string1 – cost 1
  - Insert a character in string2 – cost 1
  - Substitute a character for another – cost 1
- Transform ‘kitten’ in to ‘sitting’
  - kitten -> sitten (cost 1 – replace k with s)
  - sitten -> sittin (cost 1 - replace e with i)
  - sittin -> sitting (cost 1 – add g)
- Levenshtein distance: 3

Fuzzy Matching

- Estimated string similarity computation costs:
  - Hard on the GC (lots of temporary strings created and thrown away, use arrays if possible.
  - Levenshtein can be computed in O (kl) time, where ‘l’ is the length of the shortest string, and ‘k’ is the maximum distance.
  - Parallelisable – split the set of words to compare across n cores.
  - Can do approximately 10,000 compares per second on a standard single core laptop.

Fuzzy Matching Demo



Agenda

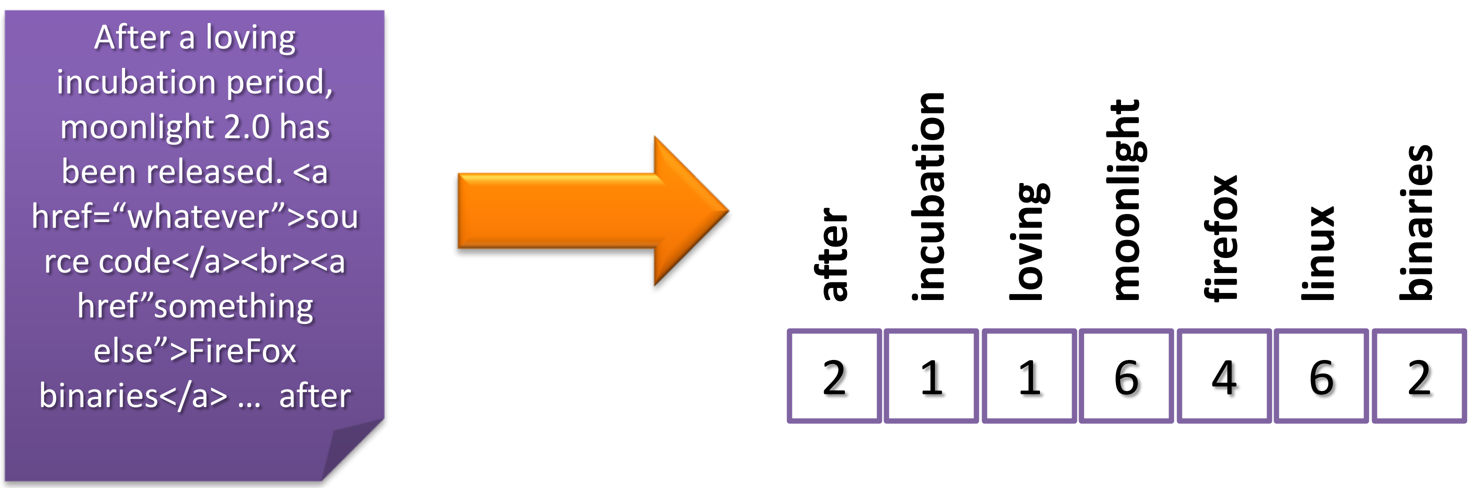
- Why Functional Programming?
- F# Language Walkthrough
- **Smart Software Ideas**
  - Recommendations
  - Fuzzy Matching
  - Search
  - Classification

Search

- Given a search term and a large document corpus, rank and return a list of the most relevant results…

Search

- Simplify:
  - For easy machine/language manipulation
  - … and most importantly, easy computation
- Vectors: natures own quality data structure
  - Convenient machine representation (lists/arrays)
  - Lots of existing vector math algorithms



Term Count

- Document1: Linux post:

the	incubation	crazy	moonlight	firefox	linux	penguin
9	1	1	6	4	6	2

- Document2: Animal post:

crazy	the	dog	penguin
2	2	1	5

- Vector space:

the	incubation	crazy	moonlight	firefox	linux	dog	penguin
9	1	1	6	4	6	0	2
2	0	2	0	0	0	1	5

Term Count Issues

the	incubation	crazy	moonlight	firefox	linux	dog	penguin
9	1	1	6	4	6	0	2
2	0	2	0	0	0	1	5

- ‘the dog penguin’
  - Linux: 9+0+2 = 11
  - Animal: 2+1+5 = 8
- ‘the’ is overweight
- Enter *TF-IDF*: Term Frequency Inverse Document Frequency
  - A weight to evaluate how important a word is to a corpus
    - i.e. if ‘the’ occurs in 98% of all documents, we shouldn’t weight it very highly in the total query

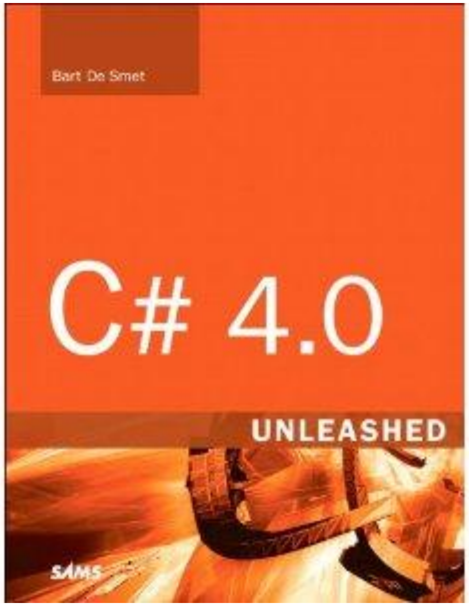
TF-IDF

- Normalise the term count:
  - tf = termCount / docWordCount

- Measure importance of term
  - idf = log (|D| / termDocumentCount)
    - where |D| is the total documents in the corpus

- tfidf = tf * idf
  - A high weight is reached by high term frequency, and a low document frequency

Search Demo



the programming language
Lua

Caml

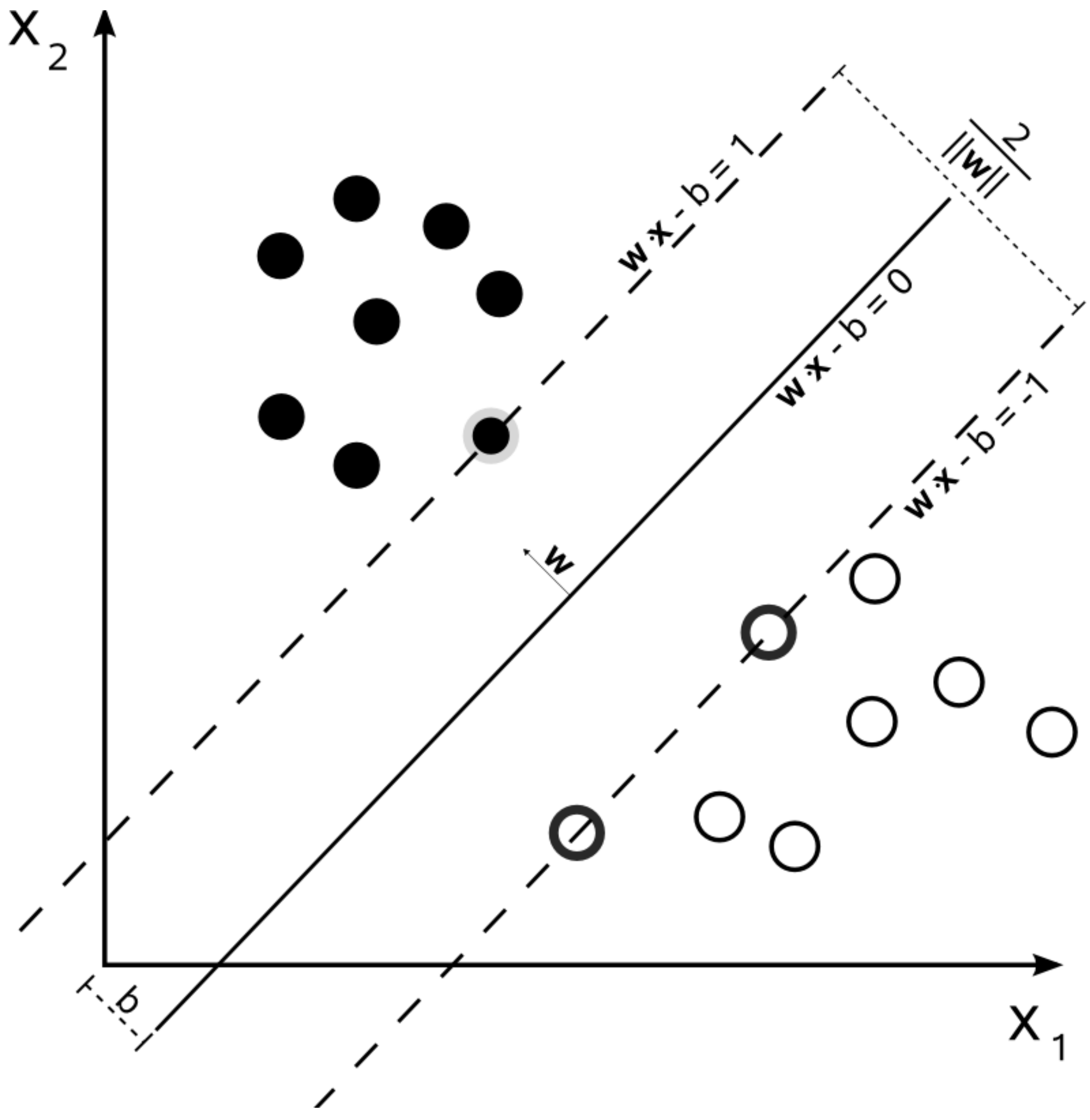
Agenda

- Why Functional Programming?
- F# Language Walkthrough
- **Smart Software Ideas**
  - Recommendations
  - Fuzzy Matching
  - Search
  - Classification

Classification

- Supervised and unsupervised methods
- Support Vector Machines (SVM)
  - Supervised learning for binary classification
  - Training Inputs: ‘in’ and ‘out’ vectors.
  - SVM will then find a separating ‘hyperplane’ in an n-dimensional space
- Training costs, but classification is cheap
- Can retrain on the fly in some cases

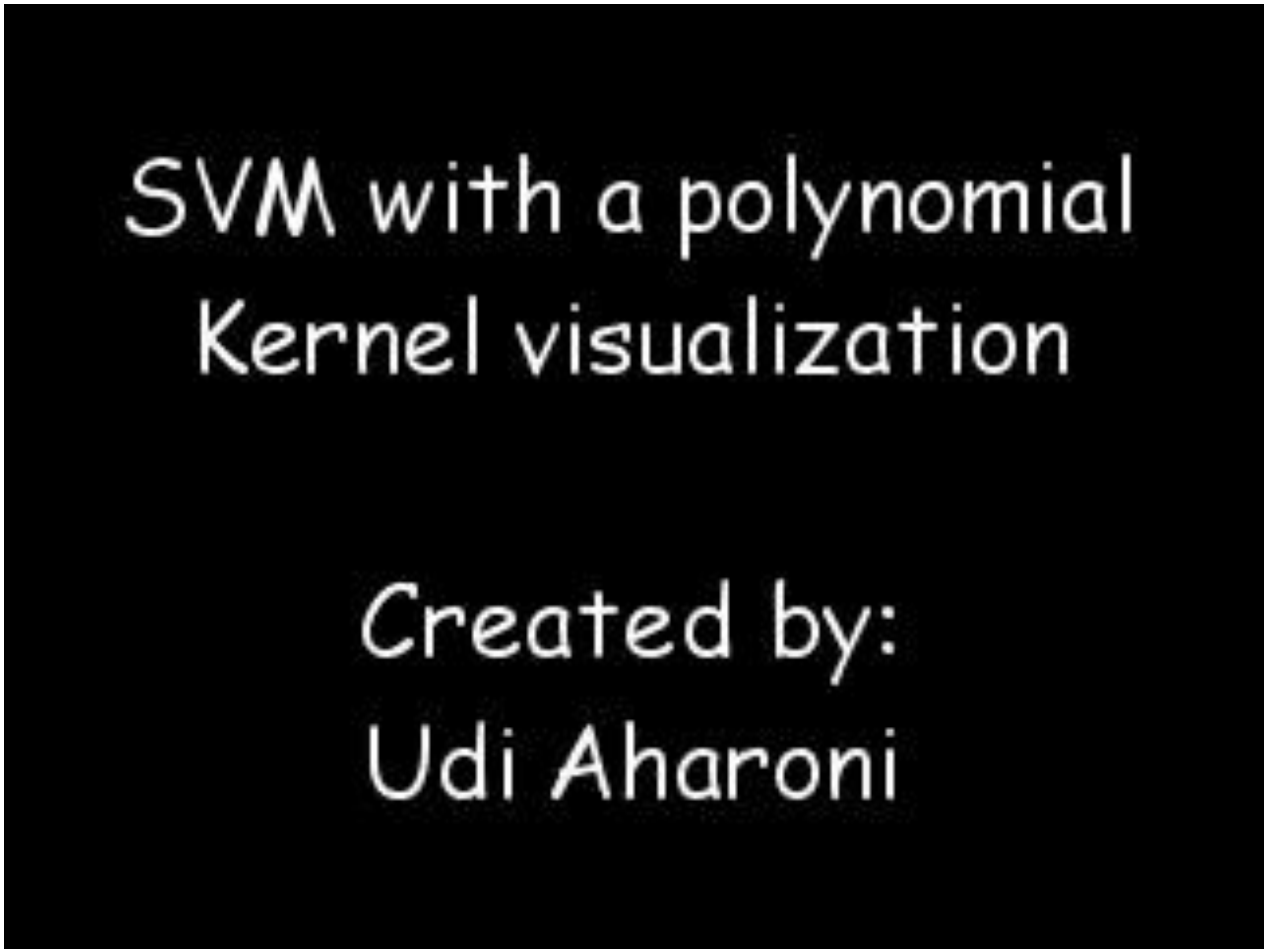
Classification



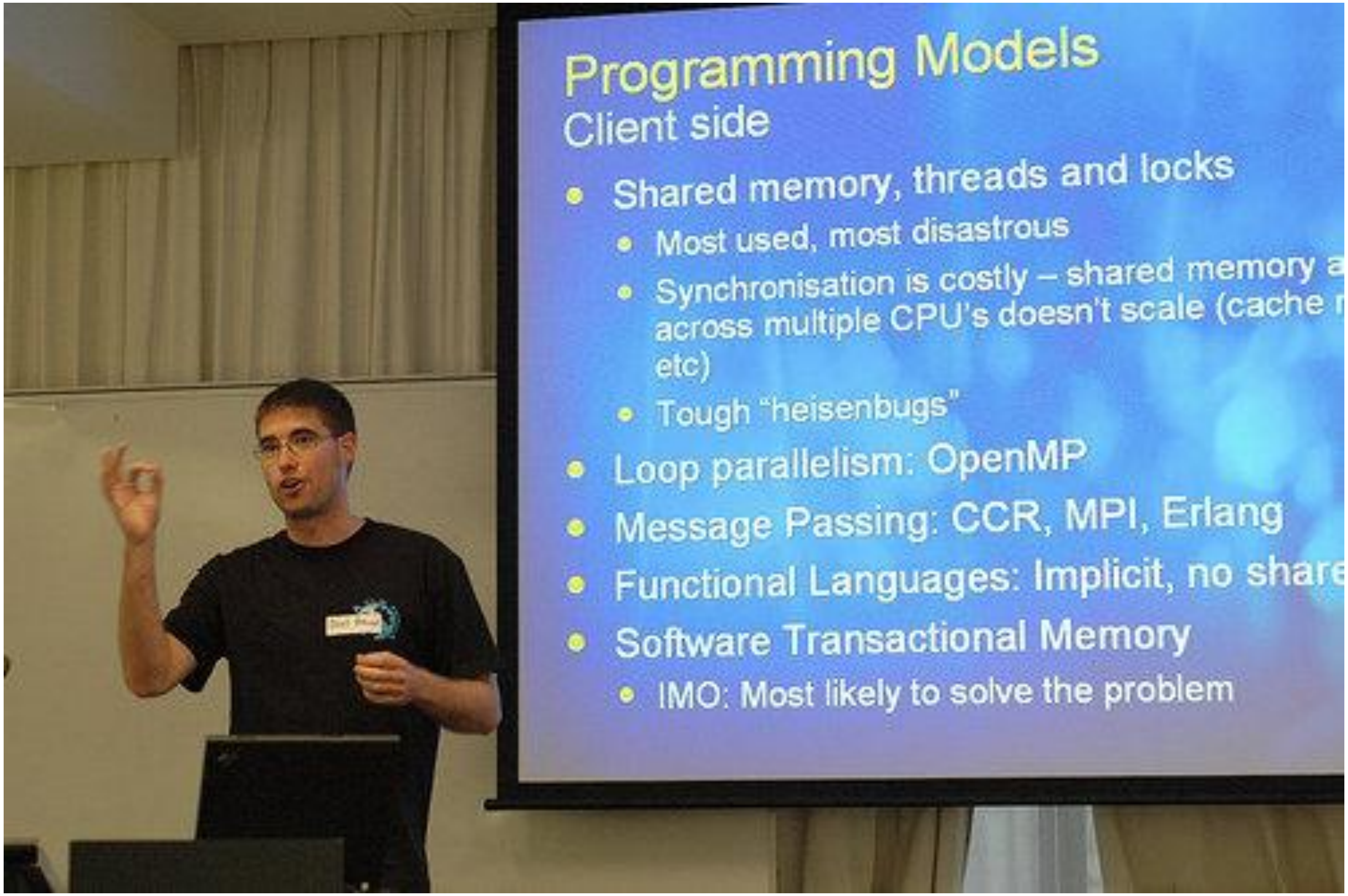
Classification

- Classification on 2 dimensions is easy, but most input is multi-dimensional
- Some ‘tricks’ are needed to transform the input data:

Classification

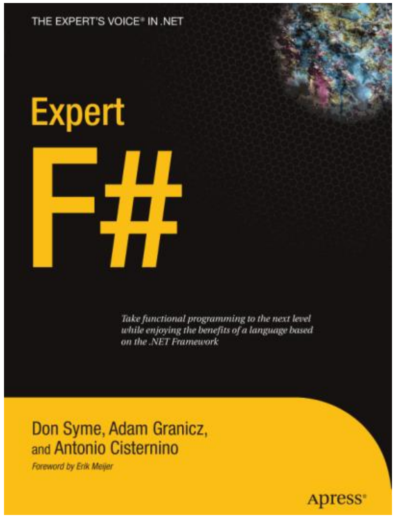


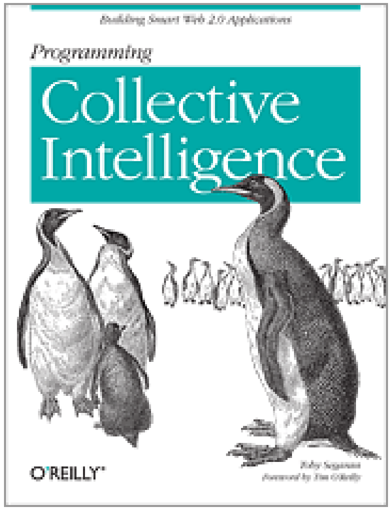
Demo: Spam Classification



Resources

- F# Developer Center
  http://fsharp.net
- hubFS
  http://cs.hubfs.net





Thanks!

- Contact: joelpobar AT gmail dot com
- Blog: http://callvirt.net/blog
- Twitter: @joelpob

Active Patterns

```
let containsVowel (word: string) =
   let letters = word.Chars
   match letters with
   | ContainsAny [‘a’; ‘e’; ‘i’; ‘o’; ‘u’] ->
true
   | _ -> false
```

Fails to compile: need a ‘when’ guard

```
   let letters = word.Chars
   match letters with
   | _ when letters.Contains(‘a’) ||
letters.Contains(‘e’) … -> true
```

Active Patterns

- Enter Active Patterns:
  - Single-Case Active Patterns
    - Converts data from one type to another. Convert from classes and values that can't be matched on, to those that can
  - Multi-Case Active Patterns
    - Partition the input space in to a known set of possible values
    - Convert input data into discriminated union type
  - Partial-Case Active Patterns
    - For data that ***doesn't always convert***
    - Return an option type

Computational Expressions

```
let read_line() = System.Console.ReadLine()
let print_string(s) = printf "%s" s

print_string "What's your name? "
let name = read_line()
print_string ("Hello, " + name)
```

```
let read_line(f) = f(System.Console.ReadLine())
let print_string(s, f) = f(printf "%s" s)

print_string("What's your name? ", fun () ->
    read_line(fun name ->
        print_string("Hello, " + name, fun () -> () ) ) )
```

Computational Expressions

```
type Result = Success of float | DivByZero

let divide x y =
       match y with
       | 0.0 -> DivByZero
       | _ -> Success (x / y)

let totalResistance r1 r2 r3 =
       let r1Res = divide 1.0 r1
       match r1Res with
       | DivByZero -> DivByZero
       | Success (x)
         ->
              let r2Res = divide 1.0 r2
              match r2Res with
              | DivByZero -> DivByZero
              | Success (x) -> ...
```

Computational Expressions

```
let totalResistance r1 r2 r3 =
  desugared.Bind(
    (divide 1.0 r1),
    (fun x ->
        desugared.Bind(
          (divide 1.0 r2),
          (fun y ->
            desugared.Bind(
              (divide 1.0 r3),
              (fun z ->
                desugared.Return(
                  divide 1.0 (x + y + z)
                  )
              ))
          ))
      )
  )
```

Computational Expressions

```
expr = ...
     | expr { cexpr }                      -- let v = expr in v.Delay(fun () -> «cexpr»)
     | { cexpr }
     | [| cexpr |]
     | [  cexpr  ]
cexpr = let! pat = expr in cexpr           -- v.Bind(expr,(fun pat -> cexpr))
      | use pat = expr in cexpr            -- v.Using(expr,(fun pat -> cexpr))
      | do! cexpr1 in cexpr2               -- let! () = cexpr1 in cexpr2»
      | do expr in cexpr                   -- let () = cexpr1 in cexpr2»
      | for pat in expr do cexpr           -- v.For(expr,(fun pat -> cexpr))
      | while expr do cexpr                -- v.While((fun () -> expr),
                                                         v.Delay(fun () -> cexpr))
      | if expr then cexpr else cexpr      -- if expr then cexpr1 else cexpr2
      | if expr then cexpr                 -- if expr then cexpr1 else v.Zero()
      | cexpr; cexpr                       -- v.Combine(cexpr1, v.Delay(fun () -> cexpr2))
      | return expr                        -- v.Return(expr)
      | yield expr                         -- v.Yield(expr)
      | match expr with [pat -> cexpr]+    -- ...
      | try cexpr finally cexpr            -- ...
      | try cexpr with pat -> cexpr        -- ...
```

Why is Multi-threading so Hard?

How Can F# Help?

- Granularity
- Purity
- Immutability
- Libraries

Not a Silver Bullet!

The State of Asynchronous I/O

```
using System;
using System.IO;
using System.Threading;
using System.Runtime.InteropServic
using System.Runtime.Remoting.Mess
using System.Security.Permissions;

public class BulkImageProcAsync
{
  public const String ImageBaseNam
  public const int numImages = 200
  public const int numPixels = 512

  public static int processImageRe

  public static int NumImagesToFin
  public static Object[] NumImages
  public static Object[] WaitObjec

  public class ImageStateObject
  {
    public byte[] pixels;
    public int imageNum;
    public FileStream fs;
  }
```

```
public static void ReadInImageCallback(IAs
{
  ImageStateObject state = (ImageStateObje
  Stream stream = state.fs;
  int bytesRead = stream.EndRead(asyncResu
  if (bytesRead != numPixels)
    throw new Exception(String.Format
        ("In ReadInImageCallback, got the
        "bytes from the image: {0}.", byte
  ProcessImage(state.pixels, state.imageNu
  stream.Close();

  FileStream fs = new FileStream(ImageBase
      ".done", FileMode.Create, FileAccess
      4096, false);
  fs.Write(state.pixels, 0, numPixels);
  fs.Close();

  state.pixels = null;
  fs = null;

  lock (NumImagesMutex)
  {
    NumImagesToFinish--;
    if (NumImagesToFinish == 0)
    {
      Monitor.Enter(WaitObject);
      Monitor.Pulse(WaitObject);
      Monitor.Exit(WaitObject);
    }
  }
}
```

```
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;

    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);
  }

  bool mustBlock = false;
  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));
}
```

"Asynchronous File I/O"

http://msdn.microsoft.com/en-us/library/kztecsys.aspx

The State of Asynchronous I/O

```
let ProcessImageAsync(i) =
  async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
          let! pixels = inStream.AsyncRead(numPixels)
          let pixels' = ProcessImage(pixels, i)
          use outStream = File.OpenWrite(sprintf "Image%d.done" i)
          do! outStream.AsyncWrite(pixels') }

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

"Asynchronous File I/O"

http://msdn.microsoft.com/en-us/library/kztecsys.aspx

Anatomy of an Async Workflow

```
let ProcessImageAsync(i) =
  async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
          let! pixels = inStream.AsyncRead(numPixels)
          let pixels' = ProcessImage(pixels, i)
          use outStream = File.OpenWrite(sprintf "Image%d.done" i)
          do! outStream.AsyncWrite(pixels') }

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

Anatomy of an Async Workflow

```
let ProcessImageAsync(i) =
  async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
          let! pixels = inStream.AsyncRead(numPixels)
          let pixels' = ProcessImage(pixels, i)
          use outStream = File.OpenWrite(sprintf "Image%d.done" i)
          do! outStream.AsyncWrite(pixels') }

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

Anatomy of an Async Workflow

```
let ProcessImageAsync(i) =
  async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
          let! pixels = inStream.AsyncRead(numPixels)
          let pixels' = ProcessImage(pixels, i)
          use outStream = File.OpenWrite(sprintf "Image%d.done" i)
          do! outStream.AsyncWrite(pixels') }

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

Anatomy of an Async Workflow

```
let ProcessImageAsync(i) =
  async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
          let! pixels = inStream.AsyncRead(numPixels)
          let pixels' = ProcessImage(pixels, i)
          use outStream = File.OpenWrite(sprintf "Image%d.done" i)
          do! outStream.AsyncWrite(pixels') }

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

Anatomy of an Async Workflow

```
let ProcessImageAsync(i) =
  async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
          let! pixels = inStream.AsyncRead(numPixels)
          let pixels' = ProcessImage(pixels, i)
          use outStream = File.OpenWrite(sprintf "Image%d.done" i)
          do! outStream.AsyncWrite(pixels') }

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

Anatomy of an Async Workflow

```
let ProcessImageAsync(i) =
  async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
          let! pixels = inStream.AsyncRead(numPixels)
          let pixels' = ProcessImage(pixels, i)
          use outStream = File.OpenWrite(sprintf "Image%d.done" i)
          do! outStream.AsyncWrite(pixels') }

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