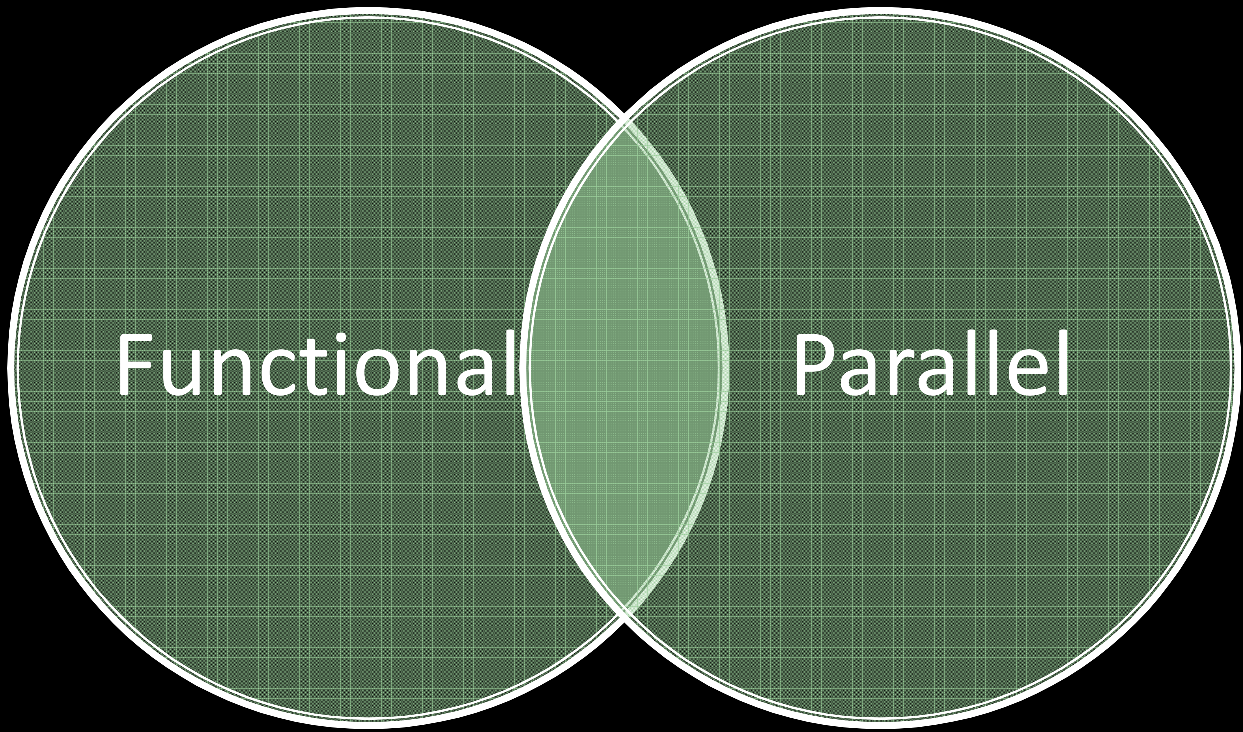
Exploring...



Don Syme
Principal Researcher
Microsoft Research, Cambridge

Disclaimer

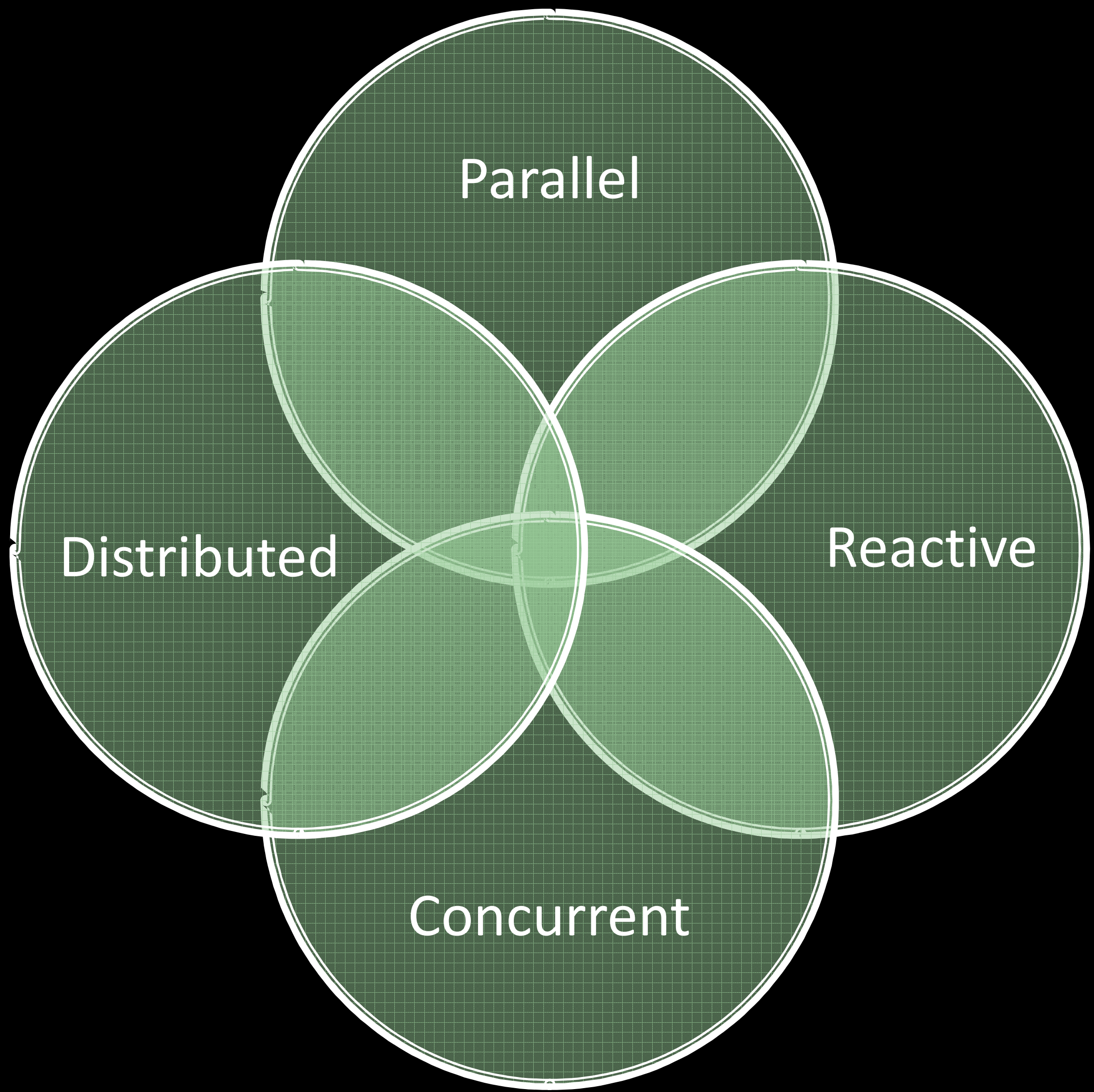
- I’m a Microsoft Guy. I’m a .NET Fan. I will be using F# and Visual Studio in this talk.
- This talk is offered in a spirit of cooperation and idea exchange. Please accept it as such ☺
- I assume “running on JVM/.NET is important”. This places some technical limitations (e.g. threads are not cheap)

Themes

- Theme: Simplicity
- Theme: Immutability
- Theme: Reaction v. Action
- Theme: Actors and Agents

Where Parallelism?

- Instruction Level (CPU)
- Multi-core Parallelism (CPUs)
- Multi-device Parallelism (CPUs+Disk)
- Multi-machine I/O Parallelism (AJAX, Client-Server)
- Multi-machine CPU Parallelism (Cluster)
- Mega-machine Parallelism (Google, Bing)
- The whole world.... (The Web!)



Parallel
Distributed
Reactive
Concurrent

some basic F#

Whitespace Matters

```
let computeDerivative f x =
    let p1 = f (x - 0.05)

  let p2 = f (x + 0.05)

       (p2 – p1) / 0.1
```

Offside (bad indentation)

Whitespace Matters

```
let computeDerivative f x =
    let p1 = f (x - 0.05)

    let p2 = f (x + 0.05)

    (p2 – p1) / 0.1
```

Functional– Pipelines

The pipeline operator

```
x |> f
```

Objects + Functional

```
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)
```

Inputs to object construction

Object internals

Exported properties

Exported method

theme: functional simplicity

```fsharp
//F#
open System
let a = 2
Console.WriteLine(a)
```

```csharp
//C#
using System;

namespace ConsoleApplication1
{
  class Program
  {
    static int a()
    {
        return 2;
    }
    static void Main(string[] args)
    {
        Console.WriteLine(a);
    }
  }
}
```

More noise than signal!

Pleasure

```
type Command = Command of (Rover -> unit)

let BreakCommand  =
  Command(fun rover -> rover.Accelerate(-1.0))

let TurnLeftCommand  =
  Command(fun rover -> rover.Rotate(-5.0<degs>))
```

Pain

```
abstract class Command
 {
     public virtual void Execute();
 }
 abstract class MarsRoverCommand : Command
 {
     protected MarsRover Rover
        { get; private set; }
     public MarsRoverCommand(MarsRover rover)
     {
        this.Rover = rover;
     }
 }
 class BreakCommand : MarsRoverCommand
 {
     public BreakCommand(MarsRover rover)
       : base(rover)
        {
        }
     public override void Execute()
        {
            Rover.Rotate(-5.0);
        }
}
class TurnLeftCommand : MarsRoverCommand
{
     public TurnLeftCommand(MarsRover rover)
```

Pleasure

```
let swap (x, y) = (y, x)


let rotations (x, y, z) =
    [ (x, y, z);
      (z, x, y);
      (y, z, x) ]


let reduce f (x, y, z) =
    f x + f y + f z
```

Pain

```
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);
}
```

Pleasure

```
type Expr =
    | True
    | And  of Expr * Expr
    | Nand of Expr * Expr
    | Or   of Expr * Expr
    | Xor  of Expr * Expr
    | Not  of Expr
```

Pain

```
public abstract class Expr { }
public abstract class UnaryOp :Expr
{
    public Expr First { get; private set; }
    public UnaryOp(Expr first)
    {
        this.First = first;
    }
}

public abstract class BinExpr : Expr
{
    public Expr First { get; private set; }
    public Expr Second { get; private set; }

    public BinExpr(Expr first, Expr second)
    {
        this.First = first;
        this.Second = second;
    }
}

public class TrueExpr : Expr { }

public class And : BinExpr
{
    public And(Expr first, Expr second) : base(fir
```

Pleasure

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

Pain

```
public abstract class Event { }

public class PriceEvent : Event
{
    public Price Price { get; private set; }
    public PriceEvent(Price price)
    {
        this.Price = price;
    }
}

public class SplitEvent : Event
{
    public double Factor { get; private set; }

    public SplitEvent(double factor)
    {
        this.Factor = factor;
    }
}

public class DividendEvent : Event
{
  ...
}
```

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

|> Async.RunSynchronously
```

```
Async.Parallel [ for i in 0 .. 200 -> computeTask i ]

|> Async.RunSynchronously
```

Taming Asynchronous I/O

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

public class BulkImageProcAsync
{
    public const String ImageBas
    public const int numImages =
    public const int numPixels =

    // ProcessImage has a simple
    // of times you repeat that
    // bound or more IO-bound.
    public static int processIma

    // Threads must decrement Nu
    // their access to it throug
    public static int NumImagesT
    public static Object[] NumIm
    // WaitObject is signalled w
    public static Object[] WaitO
    public class ImageStateObjec
    {

    }
```

```
public static void ReadInImageCallback(IAsyncRes
{
    ImageStateObject state = (ImageStateObject)a
    Stream stream = state.fs;
    int bytesRead = stream.EndRead(asyncResult);
    if (bytesRead != numPixels)
        throw new Exception(String.Format
            ("In ReadInImageCallback, got the wr
            "bytes from the image: {0}.", bytesR
    ProcessImage(state.pixels, state.imageNum);
    stream.Close();

    // Now write out the image.
    // Using asynchronous I/O here appears not t
    // It ends up swamping the threadpool, becau
    // threads are blocked on I/O requests that
    // the threadpool.
    FileStream fs = new FileStream(ImageBaseName
        ".done", FileMode.Create, FileAccess.Wri
        4096, false);
    fs.Write(state.pixels, 0, numPixels);
    fs.Close();
h memo
e is a
w.
            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;
        // 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.
    // I  t  block until all are finished.
    bool mustB
    lock (NumImagesMu
    {
        if (NumImagesToFinish >
            mustBlock = true;
    }
    if (mustBlock)
    {
        Console.WriteLine("All worker threads are queued. " +
            " Blocking until they complete. numLeft: {0}",
            NumImagesToFinish);

    }
    lon
    Con                                ing images: {0}ms",
        (t1 - t0));
  }
}
```

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

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

Processing 200 images in parallel

Equivalent F#, more robust

theme: language

Some Micro Trends

- Communication With Immutable Data
- Programming With Queries
- Programming With Lambdas
- Programming With Pattern Matching
- Languages with a Lighter Syntax
- Taming Side Effects

REST, HTML, XML, JSON, Haskell, F#, Scala, Clojure, Erlang,...

C#, VB, F#, SQL, Kx....

C#, F#, Javascript, Scala, Clojure, ...

F#, Scala, ...

Python, Ruby, F#, ...

Erlang, Scala, F#, Haskell, ...

The Huge Trends

THE WEB

MULTICORE

Myths and Fallacies

- "Multi-core changes everything"

  ✗ It changes what the web and cloud haven't changed already

- "Parallelism is all about CPU computations"

  ✗ I/O Parallelism is hugely important

- "Functional Parallelism is Implicit Parallelism"

  ✗ A lofty goal, far from reality

What does Typed Functional Programming Bring to Parallelism?

Make parallelism sane

Make I/O and task parallelism more compositional

"Minimize Mutable State"

"Purity"

Less Mutable State

"Transactional State"

"Isolated State"

"Immutable Data"

Integrate declarative engine-based parallelism into language
e.g. Queries,
Array/matrix programs,
Constraint programs

Computational Sub-Languages

"combinators"
"monads"
"comprehensions"

Declarative Sub-Languages

"meta-programs"
"DSLs"
"LINQ"

Types

"generics"

```
task { ... }
async { ... }
PSeq.map
Async.Parallel
```

Make programming sane

```
query
  <@ seq { for i in db.Customers do
           for j in db.Employees do
             if i.Country = j.Country then
               yield (i.FirstName, j.FirstName) } @>
```

theme: immutability

Immutability the norm…

- Immutable Lists
- Immutable Records
- Immutable Sets
- Immutable Objects
- Immutable Tuples
- Immutable Maps
- Immutable Unions
- + lots of language features to encourage immutability

demo

immutability

theme: react, not act

Example: F#

F# is a **Parallel** Language

(Multiple active **computations**)

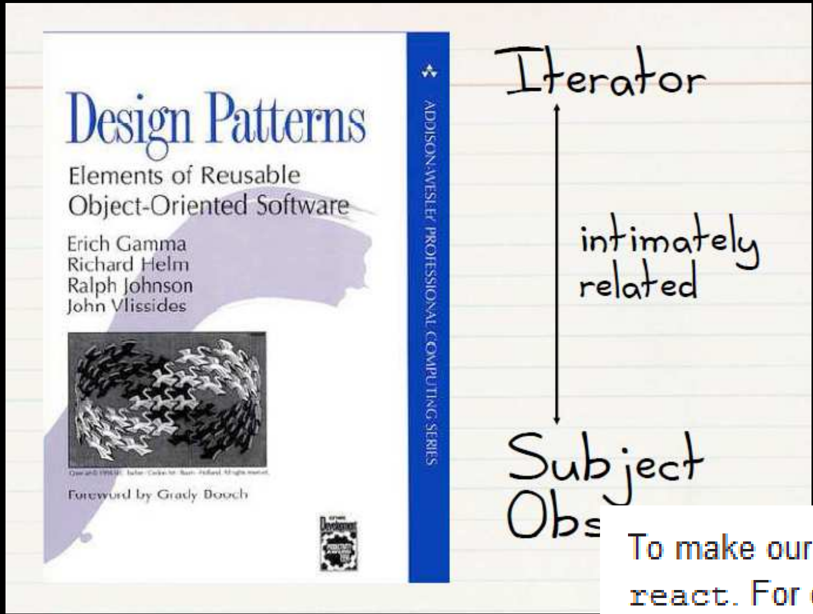
F# is a **Reactive** Language

(Multiple pending **reactions**)

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

(the same applies to Java, C#, VB, Scala, Erlang, …)

and we're still working through the ramifications of this!



F# is a **Parallel** Language

(Multiple active **computations**)

F# is a **Reactive** Language

Beyond Mere

Concurrent Functional
with Scal

Rúnar Bjarn

To make our ping and pong actors thread-less, it suffices to simply r
`react`. For example, here is the modified `act` method of our pong a

```
def act() {
  var pongCount = 0
  loop {
    react {
      case Ping =>
        if (pongCount % 1000 == 0)
          Console.println("Pong: ping "+pongCount)
        sender ! Pong
        pongCount = pongCount + 1
      case Stop =>
        Console.println("Pong: stop")
        exit()
    }
  }
}
```

Scala

```
def waitForTwoCreatures(n : int) {
    react {
        case (first : Visit) => {
            react {
            case (second : Visit) => {
                makeMatch(first, second)
                if (0 == n - 1) exit
                else waitForTwoCreatures(n - 1)
                }}}}}
```

Erlang

```
loop(MyName) ->
  receive
    { accept, SenderName, Message } ->
      io:format("~n~s receives from ~s: ~s ~n", [MyName, SenderName, Message]),
      loop(MyName);

    { say, Message } ->
      io:format("~n~s says ~s
      room ! { self(), broadca
      loop(MyName)

  end.
```

F#

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

F# async { ... }

A Building Block for Writing Reactive Code

```
async { ... }
```

async == Resumptions == One shot continuations

Example: F# async { ... }

React!

```
async { let! res = httpAsync "www.google.com"
        ...  }
```

React to a GUI Event
React to a Timer Callback
React to a Query Response
React to a HTTP Response
React to a Web Service Response
React to a Disk I/O Completion
Agent reacts to Message

Example: F# async { ... }

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

Asynchronous action

Continuation/ Event callback

You're actually writing this (approximately):

```
async.Delay(fun () ->
    async.Bind(ReadAsync "cat.jpg", (fun image ->
        let image2 = f image
        async.Bind(writeAsync "dog.jpg",(fun () ->
            printfn "done!"
            async.Return())))))
```

The many uses of F# async { ... }

- Sequencing I/O requests

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

- Sequencing CPU computations and I/O requests

```
async { let! lang  = detectLanguageAsync text
        let! text2 = translateAsync (lang,"da",text)
        let text3 = postProcess text2
        return text3 }
```

The many uses of F# async { ... }

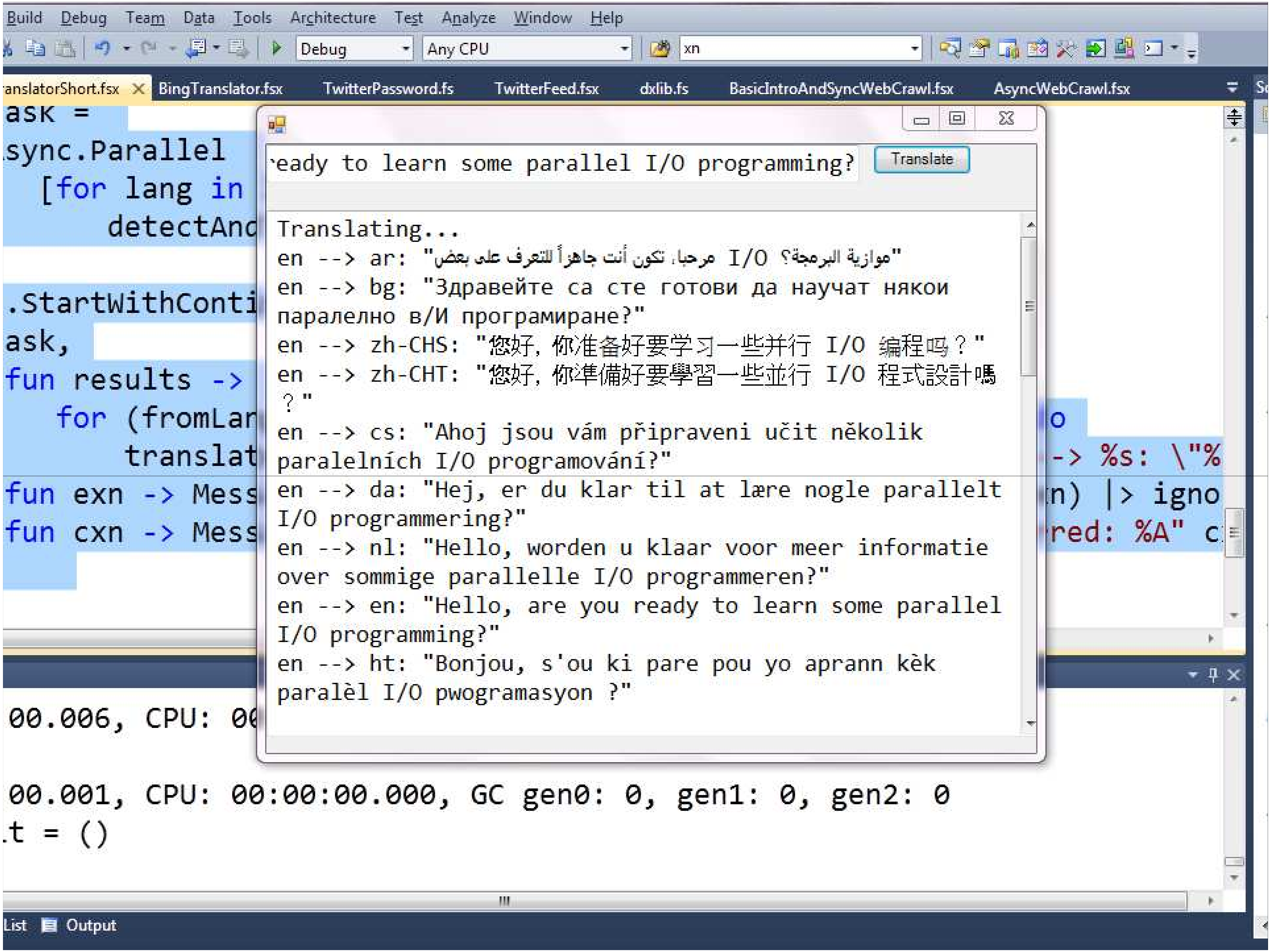
- Parallel CPU computations

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

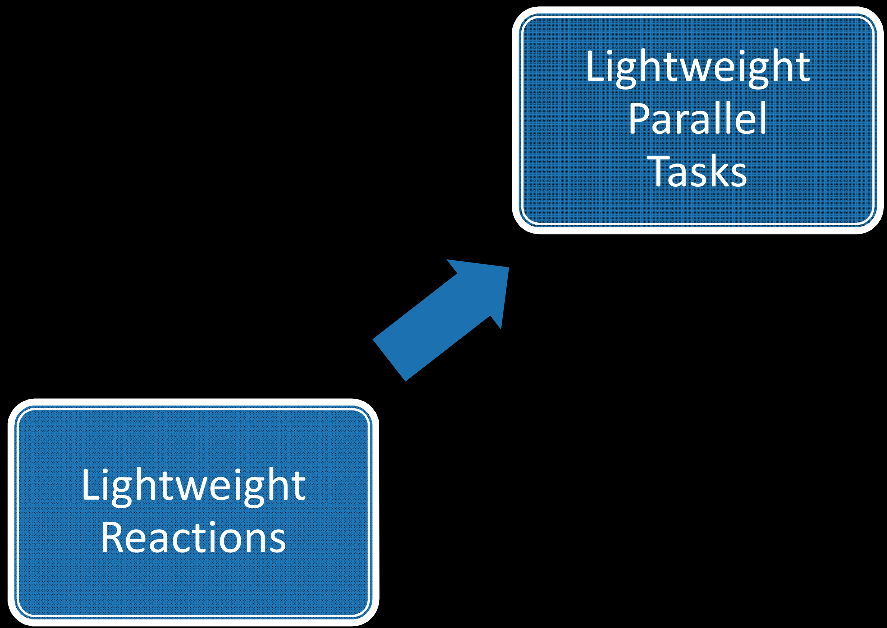
- Parallel I/O requests

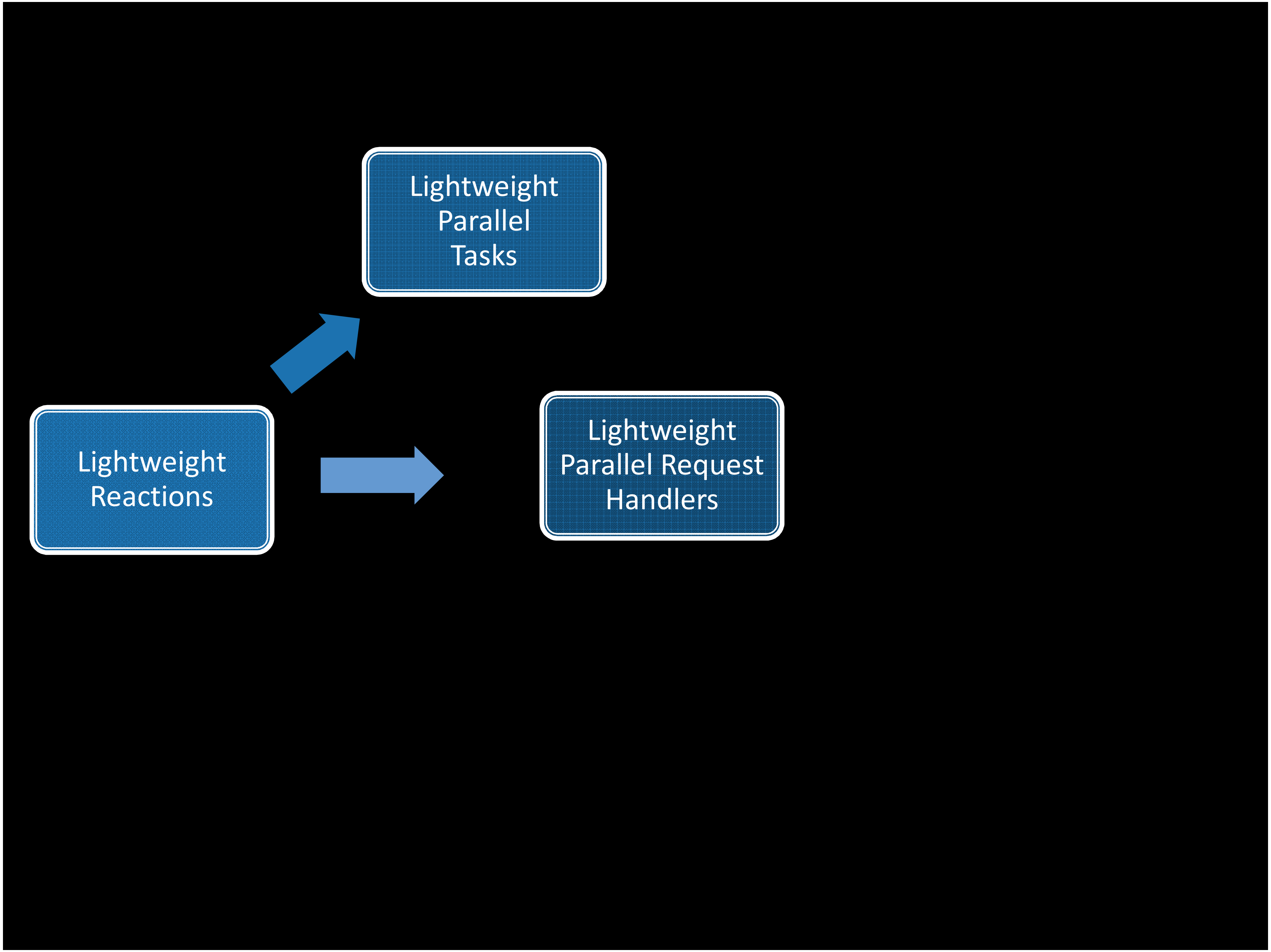
```
Async.Parallel
   [ for target in langs ->
             translateAsync (lang,target,text) ]
```

demo



Build Debug Team Data Tools Architecture Test Analyze Window Help
Debug
Any CPU
xn
anslatorShort.fsx
BingTranslator.fsx
TwitterPassword.fs
TwitterFeed.fsx
dxlib.fs
BasicIntroAndSyncWebCrawl.fsx
AsyncWebCrawl.fsx
ask =
sync.Parallel
[for lang in
detectAnd
.StartWithConti
ask,
fun results ->
for (fromLar
translat
fun exn -> Mess
fun cxn -> Mess
o
-> %s: \"%
n) |> igno
red: %A" c
·eady to learn some parallel I/O programming?
Translate
Translating...
en --> ar: "مرحبا، تكون أنت جاهزاً للتعرف على بعض I/O موازية البرمجة؟"
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，你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 --> ht: "Bonjou, s'ou ki pare pou yo aprann kèk paralèl I/O pwogramasyon ?"
00.006, CPU: 00
00.001, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
.t = ()
List
Output


Lightweight Parallel Tasks
Lightweight Reactions



Lightweight Parallel Tasks
Lightweight Reactions
Lightweight Parallel Request Handlers

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

```
/// 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
    }
```

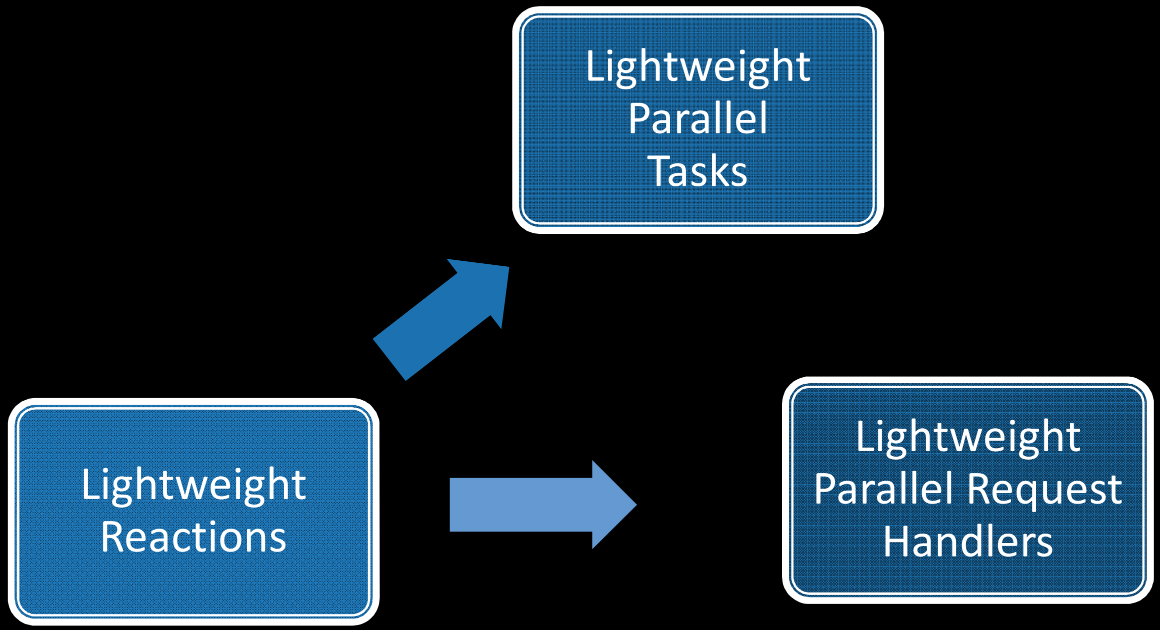
React!

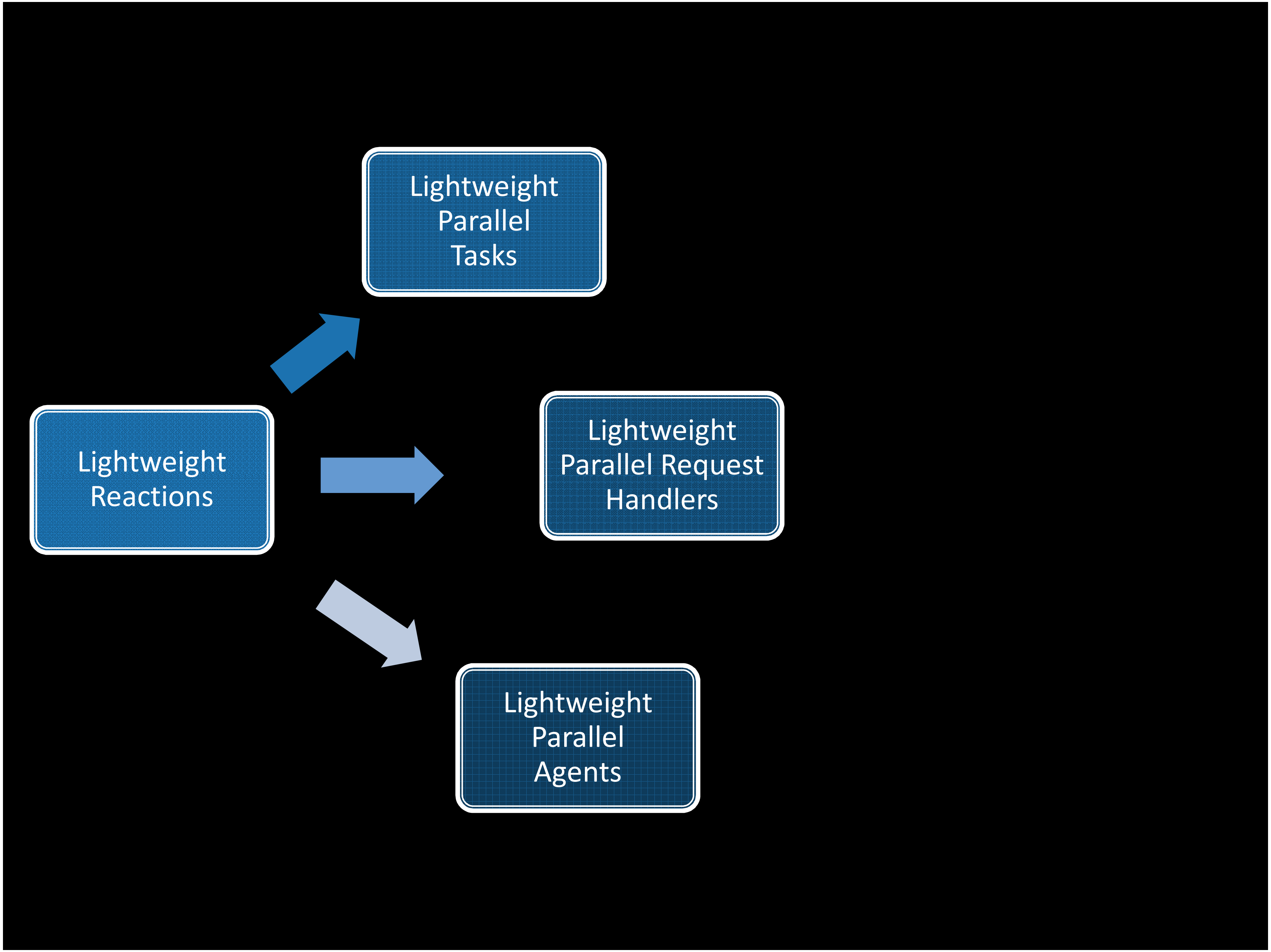
React!

React!

```
let server() =
    AsyncTcpServer(IPAddress.Loopback, 10000, handleServerRequestAsync)
```

theme: agents/actors


Lightweight
Parallel
Tasks
Lightweight
Reactions
Lightweight
Parallel Request
Handlers


Lightweight Parallel Tasks
Lightweight Reactions
Lightweight Parallel Request Handlers
Lightweight Parallel Agents

Actor concurrency

So what's the alternative? The actor model of concurre
achieved by its flagship language, Erlang.

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

Let's look at these principles in more detail. An actor is

The many uses of F# async { ... }

- Repeating tasks

```
async { let state = ...
        while true do
          let! msg = queue.ReadMessage()
          <process message> }
```

- Repeating tasks with immutable state

```
let rec loop count =
    async { let! msg = queue.ReadMessage()
            printfn "got a message"
            return! loop (count + msg) }

loop 0
```

A First Agent

```
let agent =

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

```
Note:
type Agent<'T> = MailboxProcessor<'T>
```

```
agent.Post "three"
agent.Post "four"
```

First 100,000 Agents

```
let agents =
   [ for i in 0 .. 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"
```

Note:
```
type Agent<'T> = MailboxProcessor<'T>
```

A Chatty Agent

```
let agent =
       Agent.Start(fun inbox ->
         async { while true do
                   let! a,b,resp = inbox.Receive()
                   resp.Reply (a+b) })
```

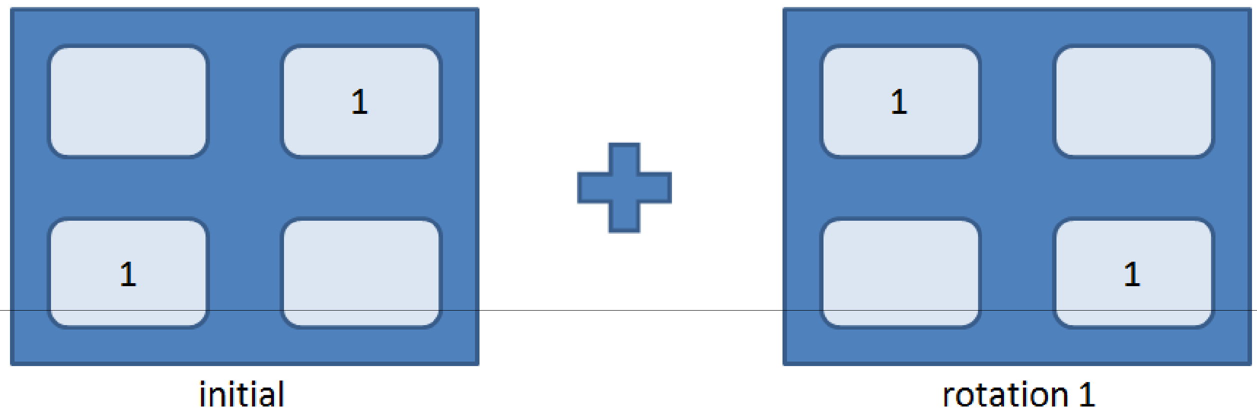
Response

```
agent.PostAndAsyncReply (fun resp -> (10,10,resp))
```

Request

demo

theme: data parallelism

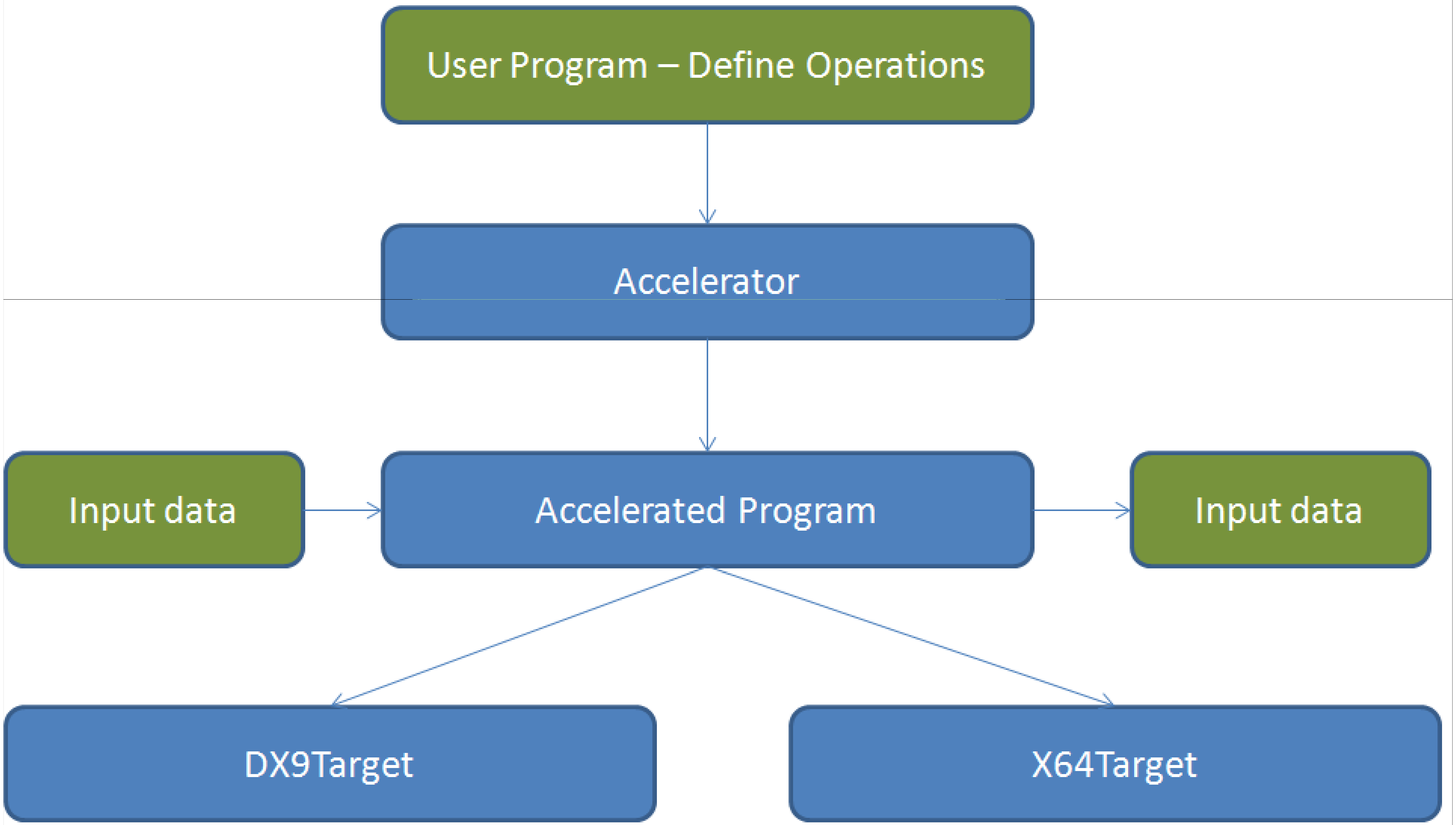


etc. ... = Sum of all neighbours

Data Parallelism: Philosophies

- Functional is beautiful for declarative data parallelism
  - Just write the equations
- But how to run it?
  - Smart, dedicated compiler (e.g. Sisal)
  - Embedded Expression-based DSL (e.g. C#/F#)
  - Compile-time Meta-programming (e.g. Haskell)
  - Run-time Meta-programming (e.g. C#/F#)

Microsoft Accelerator



Microsoft Accelerator – Code!

```
let nums = [| 6; 1; 5; 5; 3 |]
let input = new FloatParallelArray(nums)
let sum = ParallelArrays.Shift(input, 1)
          + input
          ParallelArrays.Shift(input, -1)
let output = sum / 3.0f;
```

```
let target = new DX9Target();
let res = target.ToArray1D(output);
```

Example: F# Game of Life

```
/// Evaluate next generation of the life game state

let nextGeneration (grid: Matrix<float32>) =

  // Shift in each direction, to count the neighbours
  let sum = shiftAndSum grid offsets

  // Check to see if we're born or remain alive
  (sum =. threeAlive) ||. ((sum =. twoAlive) &&. grid)
```

CPU

Example: F# Game of Life

```
/// Evaluate next generation of the life game state
[<ReflectedDefinition>]
let nextGeneration (grid: Matrix<float32>) =

  // Shift in each direction, to count the neighbours
  let sum = shiftAndSum grid offsets

  // Check to see if we're born or remain alive
  (sum =. threeAlive) ||. ((sum =. twoAlive) &&. grid)
```

CPU

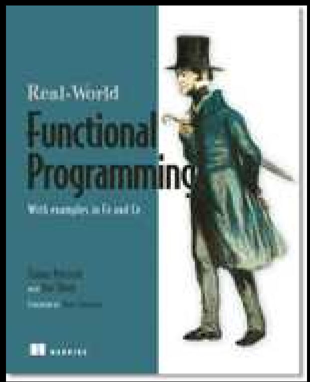
GPU

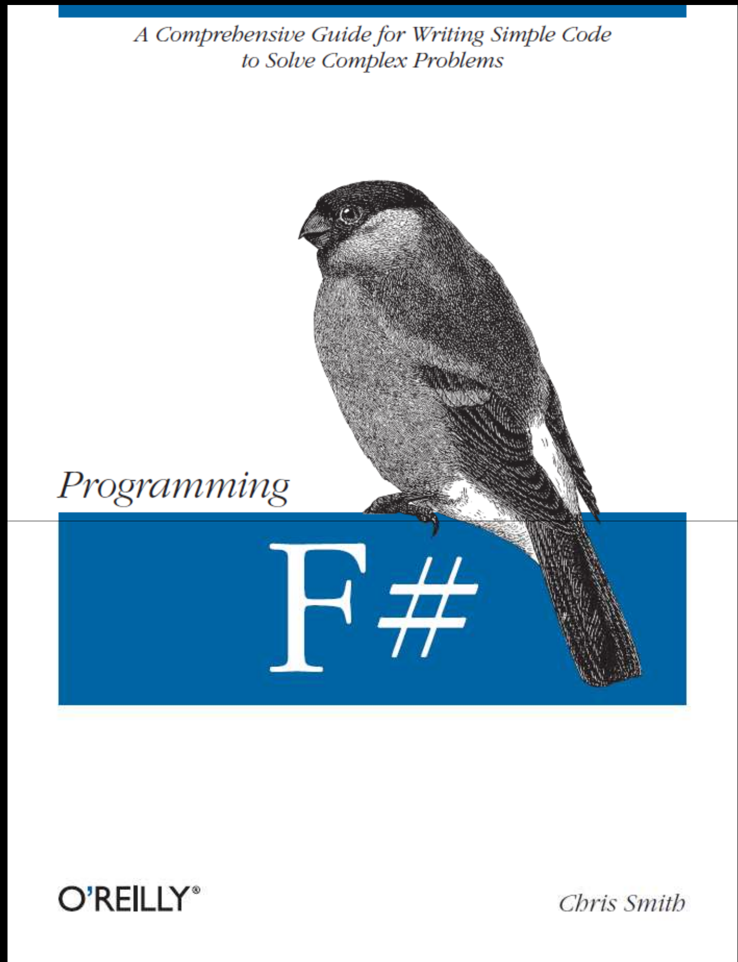
summary

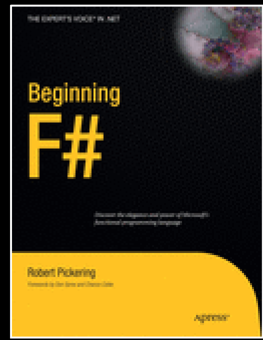
Key Themes

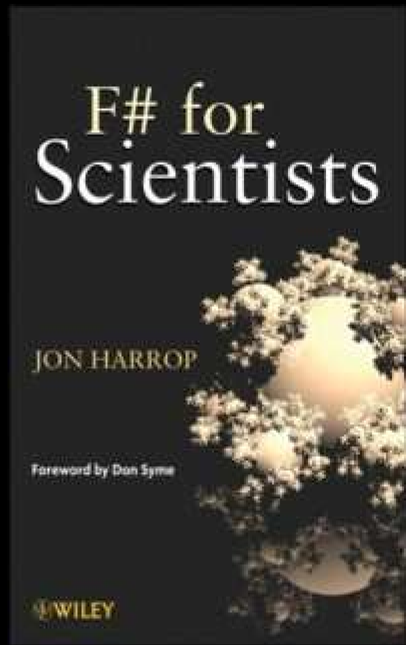
- Simplicity of Expression
- Composability
- Immutability
- Lightweight Reaction (tasks, agents, actors, promises, futures)
- Transactions
- Data Parallelism

Latest Books about F#









www.fsharp.net

question & answer