

Introduction to Concurrency in F#

Joey Dodds

F#

- F# Warmup
- F# async basics
 - async
 - let!
 - examples
- Continuations
- Events

F# Warmup

- F# Interactive environment
 - [Download here](#)

F# Parallel Needs

- F# has constructs for asynchronous, reactive programming
 - Code should be allowed to execute independent of the main thread
 - Should be able to react quickly to updates in environment...

F# Parallel Needs

- Why react quickly to updates?
 - Timely response to user input
 - Network requests could complete at any time
 - Disk access is slow
 - GUI updates should be prompt and correct

The old way

- In many programming languages we can implement reactive solutions with callbacks
- Callbacks have some problems though:
 - Complex code
 - Difficult to correctly dispose of resource
 - Problems only multiply as code gets bigger

The old way

```
let openFileCallback() =  
    let fs = new FileStream(@"C:\Program Files\...,  
                           FileMode.Open, FileAccess.Read,  
                           FileShare.Read)  
    let data = Array.create (int fs.Length) 0uy  
    let callback ar =  
        let bytesRead = fs.EndRead(ar)  
        fs.Dispose()  
        printfn "Read Bytes: %i, First bytes were: %i %  
                %i ..."  
                bytesRead data.[1] data.[2] data.[3]  
    fs.BeginRead(data, 0, data.Length,  
                (fun ar -> callback ar), null) |> ignore
```

We create a callback function that declares behavior when the read completes

The old way

```
let openFileCallback() =  
    let fs = new FileStream(@"C:\Program Files\...,  
                           FileMode.Open, FileAccess.Read,  
                           FileShare.Read)  
    let data = Array.create (int fs.Length) 0uy  
    let callback ar =  
        let bytesRead = fs.EndRead(ar)  
        fs.Dispose()  
        printfn "Read Bytes: %i, First bytes were: %i %  
                %i ..."  
                bytesRead data.[1] data.[2] data.[3]  
    fs.BeginRead(data, 0, data.Length,  
                (fun ar -> callback ar), null) |> ignore
```



We then pass the callback to BeginRead, which expects it as an argument

The old way

We would normally like to use a “use” instead of a “let” here, because it would dispose of the resource as soon as it goes out of scope.

```
let openFileCallback() =  
    let fs = new FileStream(@"C:\Program Files\...,  
                           FileMode.Open, FileAccess.Read,  
                           FileShare.Read)  
    let data = Array.create (int fs.Length) 0uy  
    let callback ar =  
        let bytesRead = fs.EndRead(ar)  
        fs.Dispose()  
        printfn "Read Bytes: %i, First bytes were: %i %  
                %i ..."  
                bytesRead data.[1] data.[2] data.[3]  
    fs.BeginRead(data, 0, data.Length,  
                (fun ar -> callback ar), null) |> ignore
```

Here we can't use “use” because fs will go out of scope at the end of the function. We have to make this call to dispose of the resource. If the program crashes, we might never dispose of it at all.

F# Async

- Async defines a block of code we would like to run asynchronously
- We can use `let!` instead of `let`
 - `let!` binds asynchronously
 - The computation in the `async` block waits until the `let!` completes
 - While it is waiting it does not block
 - Once the `let!` Binding completes, the `async` block continues
 - Might continue on a different thread

F# Async

- What do we mean when we say let! doesn't block?
 - The computation in the async stops until the binding is complete
 - No program or OS thread is blocked
 - All we need to do is register our async
 - Fairly low cost per async
 - Can have thousands of asyncs waiting without much performance cost
 - Asyncs could even improve performance on a single thread

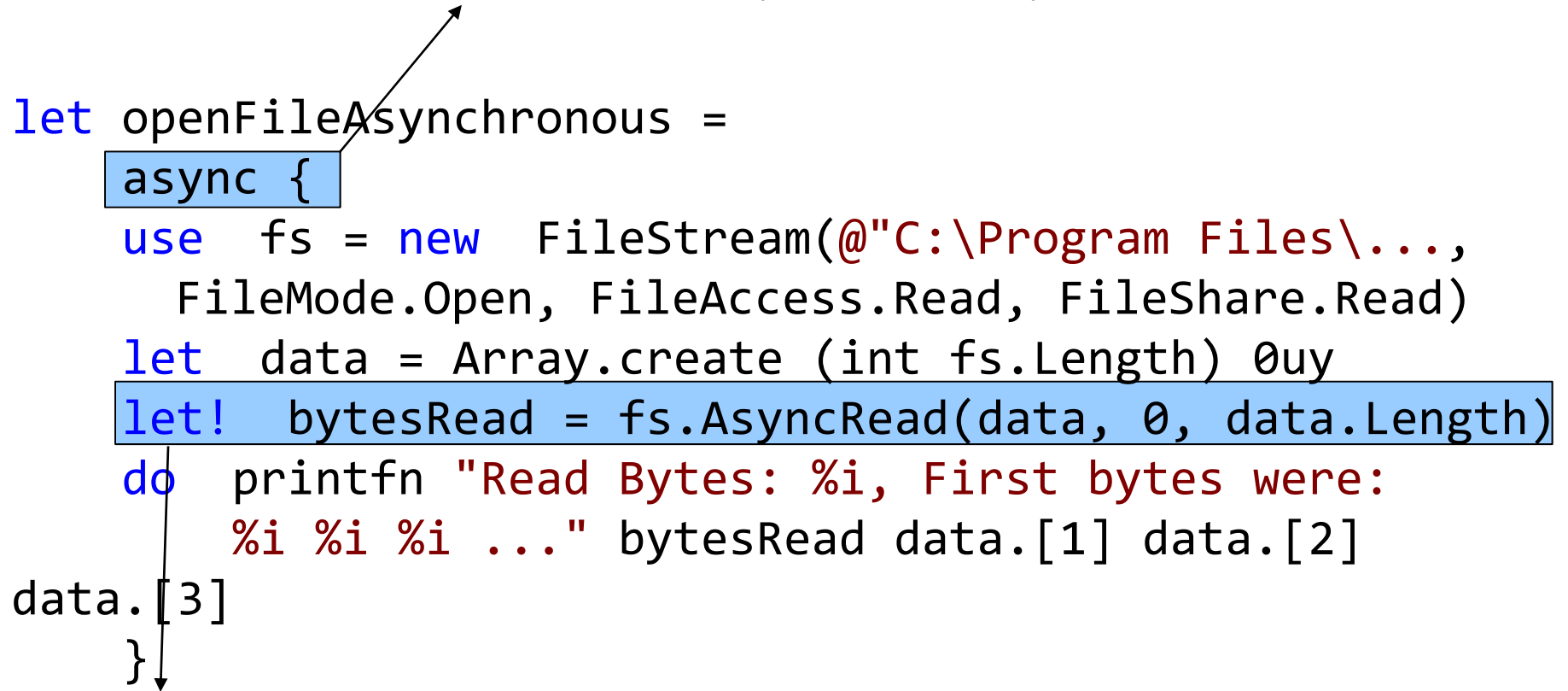
F# Async

- In some cases we can also use the “use” keyword in place of let
 - Use bindings are disposed of when they are out of scope
 - We can't use them with callbacks because they may be disposed of too soon
 - In asynchs we can use them, because we don't leave the async until our computation is done

F# Async

We use the async keyword around code we want executed asynchronously.

```
let openFileAsynchronous =  
    async {  
        use fs = new FileStream(@"C:\Program Files\...,  
                                FileMode.Open, FileAccess.Read, FileShare.Read)  
        let data = Array.create (int fs.Length) 0uy  
        let! bytesRead = fs.AsyncRead(data, 0, data.Length)  
        do printfn "Read Bytes: %i, First bytes were:  
                   %i %i %i ..." bytesRead data.[1] data.[2]  
        data.[3]  
    }
```



This binding occurs asynchronously. The computation waits here until `fs.AsyncRead` completes. The right hand side of `!` operations must be asynchronous, otherwise they would have no meaning.

Examples

- Back to the IDE for a couple of examples
 - File reading
 - Synchronous file read
 - Asynchronous file read done manually with callbacks
 - Async file read
 - Prime number finding
 - C# Style asynchronous prime number finding
 - Async prime number finding
 - Asynchronous http requests

CPU bound vs IO bound computations

- The examples fell into two categories
 - CPU Bound
 - The prime example
 - IO Bound
 - The file read example
 - The http example
- While both use the same programming constructs they have some differences

- CPU Bound functions will scale in proportion to the number of threads that can be run simultaneously
 - Running asynchronously will still occupy threads
 - On this machine we saw a 2x speedup at best
- IO bound functions can scale regardless of threads
 - IO bound computations can often “overlap”
 - This can even work for huge numbers of computations
 - We can see 100x+ speedups even on single cores

Limitations

- What we have learned so far has some limitations:
 - What do we do if our parallel code throws an exception?
 - What do we do if we need to cancel our parallel jobs?
 - How can we safely update the user with correct information once we have our results?

Continuations

- All three of those issues can be solved by running our asyncs with `Async.RunWithContinuations`.
 - Function takes 4 arguments
 - Async
 - Continuation
 - Exception Continuation
 - Cancellation Continuation

Advantages of Continuations

- Specify behavior for different error cases
- Continuations run on the same thread the async was started from
 - If you try to update the GUI from an async, you may not have access
 - With continuations you can “schedule” an update when your computation completes
 - This allows for timely and accurate GUI updates

Advantages of Continuations

```
let async1 (label:System.Windows.Forms.Label) filename =  
    Async.StartWithContinuations(  
        async {  
            label.Text <- "Operation started."  
            use outputFile =  
System.IO.File.Create(filename)  
            do! outputFile.AsyncWrite(bufferData)  
        },  
        (fun _ -> label.Text <- "Operation completed."),  
        (fun _ -> label.Text <- "Operation failed."),  
        (fun _ -> label.Text <- "Operation canceled."))
```

These three functions determine the behavior when the async completes. If we named the argument we could use it as well. For the first we could use the information computed in the async. The second and third could analyze the reason for failure/cancellation.

More limitations

- Continuations give us more power but:
 - Can't update progress (events)
 - Only know how to perform a fixed number of jobs
 - We can't generate new jobs as work progresses
 - Progress of asyncs generated by `Async.Parallel` can't be directed (agents)

Events

- We can use events to allow asyncs to report progress
- The simplest form of this is reporting when each individual job completes

Worker

```
type AsyncWorker<'T>(jobs: seq<Async<'T>>) =  
    let jobCompleted = new Event<int * 'T>()  
  
    member x.Start() =  
        let syncContext = SynchronizationContext.CaptureCurrent()  
  
        let jobs = jobs |> Seq.mapi (fun i job -> (job,i+1))  
        let work =  
            Async.Parallel  
                [ for (job,jobNumber) in jobs ->  
                    async { let! result = job  
                        syncContext.RaiseEvent jobCompleted  
                            (jobNumber,result)  
                        return result } ]  
  
            Async.Start(work |> Async.Ignore)  
  
    member x.JobCompleted = jobCompleted.Publish
```

We are going to create a worker that can report when it is done with a task.

Worker

```
type AsyncWorker<'T>(jobs: seq<Async<'T>>) =  
    let jobCompleted = new Event<int * 'T>()  
  
    member x.Start() =  
        let syncContext = SynchronizationContext.CaptureCurrent()  
  
        let jobs = jobs |> Seq.mapi (fun i job -> (job,i+1))  
        let work =  
            Async.Parallel  
                [ for (job,jobNumber) in jobs ->  
                    async { let! result = job  
                        syncContext.RaiseEvent jobCompleted  
                            (jobNumber,result)  
                        return result } ]  
  
            Async.Start(work |> Async.Ignore)  
  
    member x.JobCompleted = jobCompleted.Publish
```

First we create an event that we can use

In the start function, we have to grab the current context so we know where to raise our event.

Worker

```
type AsyncWorker<'T>(jobs: seq<Async<'T>>) =  
    let jobCompleted = new Event<int * 'T>()  
  
    member x.Start() =  
        let syncContext = SynchronizationContext.CaptureCurrent()  
  
        let jobs = jobs |> Seq.mapi (fun i job -> (job,i+1))  
        let work =  
            Async.Parallel  
                [ for (job,jobNumber) in jobs ->  
                  async { let! result = job  
                          syncContext.RaiseEvent jobCompleted  
                          (jobNumber,result)  
                          return result } ]  
  
        Async.Start(work |> Async.Ignore)  
  
    member x.JobCompleted = jobCompleted.Publish
```

We number off our jobs first and create and compose some asyncs in the way we expect

Worker

```
type AsyncWorker<'T>(jobs: seq<Async<'T>>) =  
    let jobCompleted = new Event<int * 'T>()  
  
    member x.Start() =  
        let syncContext = SynchronizationContext.CaptureCurrent()  
  
        let jobs = jobs |> Seq.mapi (fun i job -> (job,i+1))  
        let work =  
            Async.Parallel  
                [ for (job,jobNumber) in jobs ->  
                  async { let! result = job  
                          syncContext.RaiseEvent jobCompleted  
                            (jobNumber,result)  
                          return result } ]  
  
            Async.Start(work |> Async.Ignore)  
  
    member x.JobCompleted = jobCompleted.Publish
```

When a job completes, we continue in the async, raising an event in the context we started the async from. The event is aware of the job number and the result

Worker

```
type AsyncWorker<'T>(jobs: seq<Async<'T>>) =  
    let jobCompleted = new Event<int * 'T>()  
  
    member x.Start() =  
        let syncContext = SynchronizationContext.CaptureCurrent()  
  
        let jobs = jobs |> Seq.mapi (fun i job -> (job,i+1))  
        let work =  
            Async.Parallel  
                [ for (job,jobNumber) in jobs ->  
                    async { let! result = job  
                        syncContext.RaiseEvent jobCompleted  
                            (jobNumber,result)  
                        return result } ]  
  
            Async.Start(work |> Async.Ignore)  
  
    member x.JobCompleted = jobCompleted.Publish
```

As an added bonus of this style, we have encapsulated all of the asynchronous behavior in this function, so the rest of the program doesn't need to be aware of it.

Examples

- Previous example used on IO and CPU bound computations
- Worker that can report more about the computation it runs

The End

Following slides are from the draft of this
presentation

F# Warmup

- `|>` operator is a unique feature that helps write readable F# code
- To multiply a list of numbers by two and add them together
- How are we used to doing this from scratch if we already have the function below?

```
let mult2 x = 2 * x
```

```
let rec mult2add' l =  
  match l with  
  | [] -> 0  
  | h::t -> (mult2 h) + mult2add' t
```


|> Operator

```
let rec mult2add' l =  
  match l with  
  | [] -> 0  
  | h::t -> (mult2 h) + mult2add' t
```

- How would we write the same function if we had a map function?

```
let mult2add'' l =  
  List.sum (List.map mult2 l)
```

- Or using the F# |> (pipeline) operator

```
let mult2add l =  
  l  
  |> List.map mult2  
  |> List.sum
```

>> Operator

- Related to the |> operator is the >> (compose) operator
- We can use this operator when we wish to flow from one function to another without initial arguments
- For example:

```
let mult2add''' =  
    List.map mult2 >> List.sum
```

Lets do something in parallel!

- Our input will be a list of integers
- Output is a list of integer boolean pairs (number, isprime)
- We are using a silly slow implementation of IsPrime to help show the speedup
- run for i in 10000000..10004000 it takes my computer about 12-13 seconds
- the executing code is:

```
ResetStopWatch()
```

```
// primeInfo = array<int * bool>  
let primeInfo =  
    nums  
    |> Array.map (fun x -> (x, IsPrime x))
```

```
ShowTime()
```

Lets do something in parallel!

- If we know something about ThreadPool APIs we can make a good start:

```
// we need to "join" at the end to know when we're done
//, and these will help do that
let mutable numRemainingComputations = nums.Length
let mre = new ManualResetEvent(false)
// primeInfo = array<int * bool>
let primeInfo' = Array.create nums.Length (0,false)

nums
    |> Array.iteri
        (fun i x -> ignore (ThreadPool.QueueUserWorkItem(fun o ->
            primeInfo'[i] <- (x, IsPrime x)
            // if we're the last one, signal that we're done
            if Interlocked.Decrement(&numRemainingComputations) = 0 then
                mre.Set() |> ignore)))
        // wait until all done
mre.WaitOne()
ShowTime()
```

Lets do something in parallel!

- Now it runs in about 6 seconds on my dual core machine
- Great, but the code is far from beautiful

```
// we need to "join" at the end to know when we're done
//, and these will help do that
let mutable numRemainingComputations = nums.Length
let mre = new ManualResetEvent(false)
// primeInfo = array<int * bool>
let primeInfo' = Array.create nums.Length (0,false)

nums
    |> Array.iteri
        (fun i x -> ignore (ThreadPool.QueueUserWorkItem(fun o ->
            primeInfo'[i] <- (x, IsPrime x)
            // if we're the last one, signal that we're done
            if Interlocked.Decrement(&numRemainingComputations) = 0 then
                mre.Set() |> ignore)))
        // wait until all done
mre.WaitOne()
ShowTime()
```

Async

- Lets look at our original code

```
ResetStopWatch()

// primeInfo = array<int * bool>
let primeInfo =
    nums
    |> Array.map (fun x -> (x, IsPrime x))

ShowTime()
```

- And try out the F# async keyword...

```
ResetStopWatch()

let primeInfo'' =
    nums
    |> Array.map (fun x -> async { return (x, IsPrime x) } )
    |> Async.Parallel
    |> Async.RunSynchronously

ShowTime()
```


Async

- Same performance increase as the ugly version!

```
ResetStopWatch()
```

```
let primeInfo'' =  
    nums  
    |> Array.map (fun x -> async { return (x, IsPrime x) } )  
    |> Async.Parallel  
    |> Async.RunSynchronously
```

```
ShowTime()
```

- There is also a library option that gives the same performance:

```
let primeInfo''' =  
    nums  
    |> Array.Parallel.map (fun x -> (x, IsPrime x))
```

What did we just do?

- When we use `async { }` we are creating objects of the type `Async<'a>`
 - This is just a type that represents an asynchronous computation with a result of type `'a`
- `Array.map` is actually creating an `Async` for each member of our array
- `Async.Parallel` has type `seq<Async<'a>> -> Async<'a []>`
- `Async.RunSynchronously` is typed `Async<'a> -> 'a`

```
let primeInfo'' =  
    nums  
    |> Array.map (fun x -> async { return (x, IsPrime x) } )  
    |> Async.Parallel  
    |> Async.RunSynchronously
```


Some notes

- the `async` keyword isn't the only way to create the `Async` type
 - Library calls like `Stream.AsyncRead`
 - `Begin` and `End` method calls
- The `RunSynchronously` function isn't the only way to execute parallel code (more on that later)

```
let primeInfo'' =  
    nums  
    |> Array.map (fun x -> async { return (x, IsPrime x) } )  
    |> Async.Parallel  
    |> Async.RunSynchronously
```

A slightly different example

- The previous example was CPU-bound
 - Once a task started in a thread, it (probably) occupied that thread throughout the course of its computation
- We can also do I/O in parallel
- A great (and very common) example is making requests to web servers
- The goal is to request data from multiple web servers
 - We wish to do this asynchronously
 - We would also like to be able to react to the responses
 - And return results to the user in some way

A slightly different example

- The previous example was CPU-bound
 - Once a task started in a thread, it (probably) occupied that thread throughout the course of its computation
- We can also do I/O in parallel
- A great (and very common) example is making requests to web servers
- The goal is to request data from multiple web servers
 - We wish to do this asynchronously
 - We would also like to be able to react to the responses
 - And return results to the user in some way

The code

- Here is the code to make requests to multiple HTTP servers
- Notice anything new here?

```
let http url =  
    async { let req = WebRequest.Create(Uri url)  
            use! resp = req.AsyncGetResponse()  
            use stream = resp.GetResponseStream()  
            use reader = new StreamReader(stream)  
            let contents = reader.ReadToEnd()  
            return contents }  
  
let sites = ["http://www.bing.com";  
            "http://www.google.com";  
            "http://www.yahoo.com";  
            "http://www.search.com"]  
  
let htmlOfSites =  
    Async.Parallel [for site in sites -> http site ]  
    |> Async.RunSynchronously
```

The code

- Not only do we have the `use` keyword, but one of them has `!`
- `Use` simply discards the value binding as soon as it is out of scope
- The `!` has an interesting meaning in `async`
 - It tells the `async` to wait to proceed until it receives the response
 - The important thing here is that **no thread is blocked**
 - Only active computations occupy threads
 - Waiting is cheap, so we can wait a lot without slowing things down

```
let http url =  
    async { let req = WebRequest.Create(Uri url)  
            use! resp = req.AsyncGetResponse()  
            use stream = resp.GetResponseStream()  
            use reader = new StreamReader(stream)  
            let contents = reader.ReadToEnd()  
            return contents }
```

F# Scalability

- Because we never block while we wait and it doesn't cost much to wait:
 - Asynchronous I/O can scale extremely well
 - You can easily have thousands or millions of pending reactions
 - This can be useful in GUIs or web crawlers

```
let http url =  
    async { let req = WebRequest.Create(Uri url)  
            use! resp = req.AsyncGetResponse()  
            use stream = resp.GetResponseStream()  
            use reader = new StreamReader(stream)  
            let contents = reader.ReadToEnd()  
            return contents }
```

Async.StartWithContinuations

- If we want more control over what happens when our async completes (or fails) we use Async.StartWithContinuations
- This function is used in place of Async.RunSynchronously
- It has usage: Async.StartWithContinuations (
 - computation, : the asynchronous computation
 - continuation, : the function to call on completion
 - exceptionContinuation, : the function to call on exception
 - cancellationContinuation) : the function to call on cancellation
- It has the added advantage that if it starts on the GUI thread, the continuations will be called on the GUI thread.
- This means it is always safe to update the results if we have them

Using events to report progress

- StartWithContinuations was nice for reporting more information about our calculations
- It doesn't allow us to send much information during computation
- We are going to use an example that reports as soon as an individual computation finishes
- The first step is to create a type for a worker
- Our worker has:
 - an event
 - a start function
 - a function to publish events

Using events to report progress

- Our worker's start function accesses a `SynchronizationContext`
- This can basically be seen as a handle that allows asynchronous computations to run code or raise events in the correct place
- more...

Agents?

- 3-5 slides

Demo interactive environment? Earlier?
With some of the examples?