Building your perfect World with concurrent combinators in F#

RICCARDO TERRELL

Building your perfect World with concurrent combinators in F#

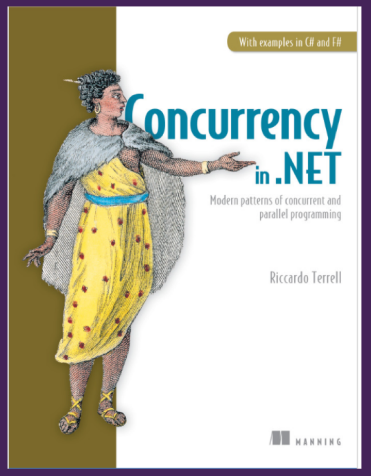
RICCARDO TERRELL

Objectives

- Implement a series of combinators to simplify concurrent programming model
- How to encapsulate Asynchronous operations
- Composition to extreme
- Solve the problem with Future (thread/async) inversion of control
- Starting with a problem, find a solution and generalize the design

Introduction - Riccardo Terrell

- Originally from Italy, currently - Living/working in Washington DC
- +/- 20 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- Author of the book “Concurrency in .NET” – Manning Pub.
- *Polyglot programmer - believes in the art of finding the right tool for the job*
- *Organizer of the Pure Functional and DC F# User Group*



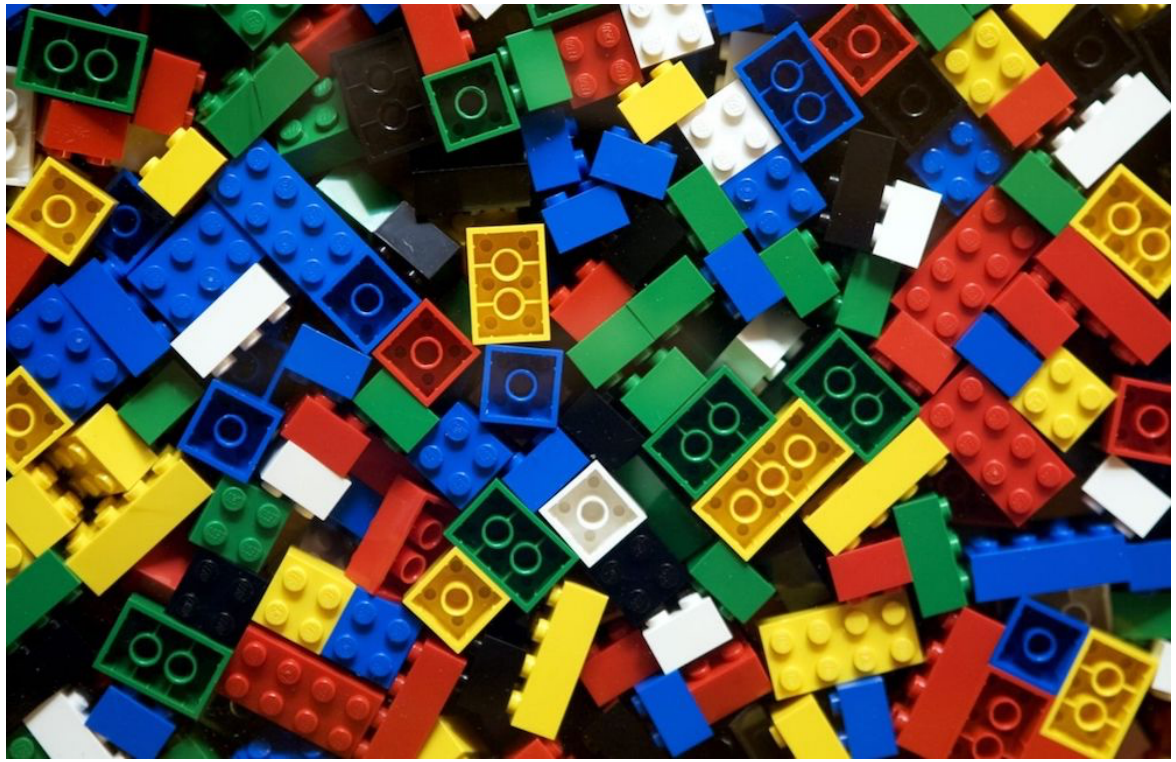
@trikace *www.rickyterrell.com* *tericcardo@gmail.com*

Combinators What's the value

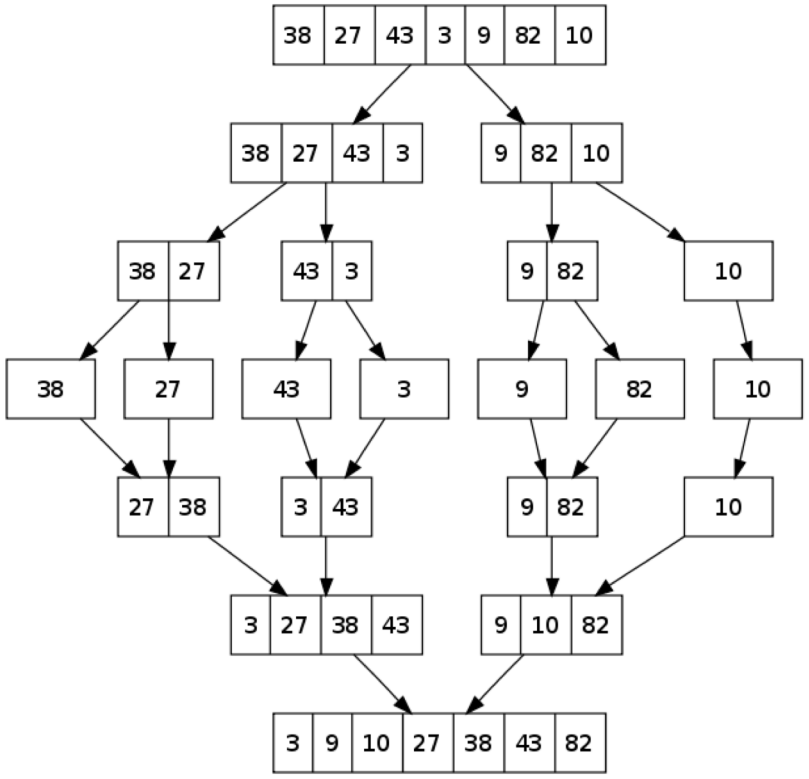
Strategies to parallelize the code

The first step in designing any parallelized system is **Decomposition**.

Decomposition is nothing more than taking a problem space and breaking it into discrete parts. When we want to work in parallel, we need to have at least two separate things that we are trying to run. We do this by taking our problem and decomposing it into parts.



Divide and Conquer




```
let rec quicksortParallelWithDepth depth aList =
    match aList with
    | [] -> []
    | firstElement :: restOfList ->
      let smaller, larger =
         List.partition (fun number -> number > firstElement) restOfList
      let left  = quicksortParallelWithDepth depth smaller
      let right = quicksortParallelWithDepth depth larger
      left @ (firstElement :: right)
```

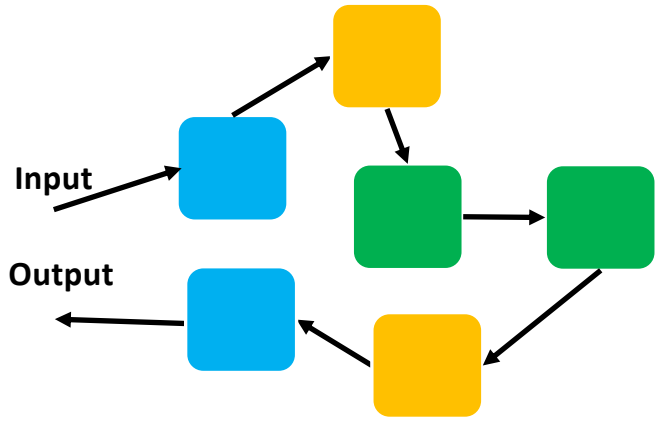
Quick Sort – with depth

```
let rec quicksortParallelWithDepth depth aList =
    match aList with
    | [] -> []
    | firstElement :: restOfList ->
        let smaller, larger =
            List.partition (fun number -> number > firstElement) restOfList
        if depth < 0 then
            let left  = quicksortParallelWithDepth depth smaller
            let right = quicksortParallelWithDepth depth larger
            left @ (firstElement :: right)
        else
            let left  = Task.Run(fun () -> quicksortParallelWithDepth (depth - 1) smaller
            let right = Task.Run(fun () -> quicksortParallelWithDepth (depth - 1) larger)
            left.Result @ (firstElement :: right.Result)
```

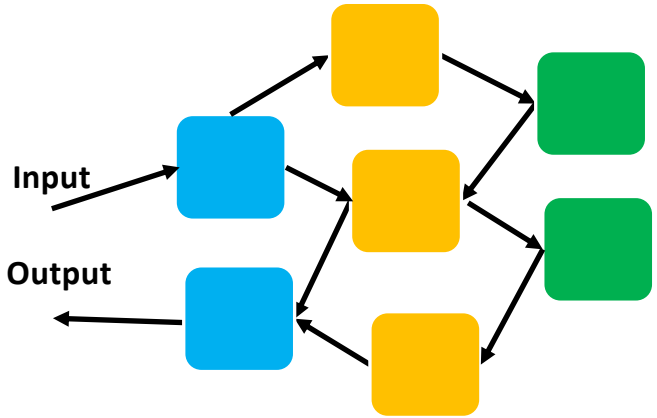
Composition and Abstraction are the Essence of Programming

Different type of concurrency models lead to different challenges

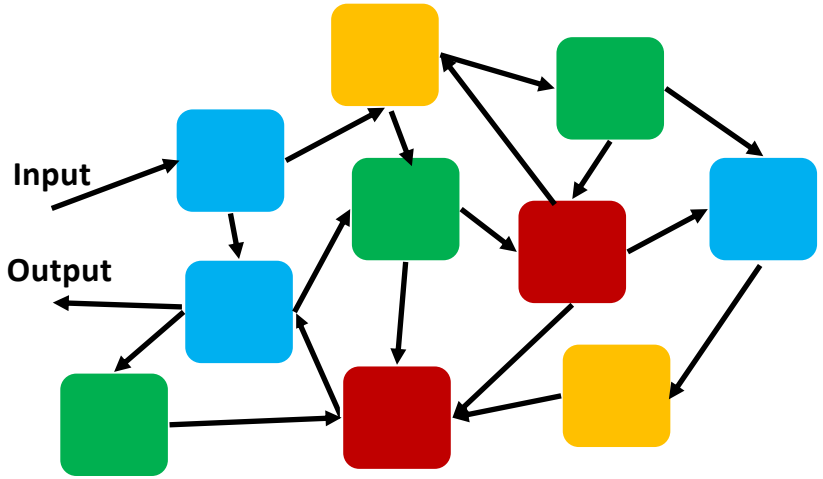
Sequential Programming



Task-Based Programming



Message-Passing Programming



Built in Combinators

Concurrency primitives
few built in combinators in .NET Core

	C# and .NET Core	F#	Shared
Data-parallel	Task.WhenAll Task.WhenAny	Async.Parallel Async.Choose (custom)	✓
Create Task	Task.Factory.Start	Async.StartChild	✓
Continuations	async/await	let!, do!	

Built-in concurrent combinators Task.WhenAll – rendezvous point

```
let a() = Task.Run(fun () -> task { Task.Delay 1000;  return 1 })
let b() = Task.Run(fun () -> task { Task.Delay 1000;  return 3 })
let c() = Task.Run(fun () -> task { Task.Delay 1000;  return 5 })

// Option 1
let result = task {
    let! a = a()
    let! b = b()
    let! c = c()
    return a + b + c
  }



// Option 2
let result = task {
    let! values = Task.WhenAll(a(), b(), c())
    return values |> Seq.sum
}
```

Task.WhenAny – fine control

Redundancy: Doing an operation multiple times and selecting the one that completes first

Interleaving: Launching multiple operations and needing them all to complete, but processing them as they complete

Throttling: Allowing additional operations to begin as others complete

Early bailout: An operation represented by t1 can be grouped in a WhenAny with another task t2, and we can wait on the WhenAny task. t2 could represent a timeout, or cancellation, or some other signal that will cause the WhenAny task to complete prior to t1 completing

Task.WhenAny - Redundancy

```
let cts = new CancellationTokenSource()

let symbol = “APPL”

let recommendations = [
    getBuyRecommendation1Async symbol cts.Token
    getBuyRecommendation2Async symbol cts.Token
    getBuyRecommendation3Async symbol cts.Token
]

let selectReccomandation = task {
    let! recommendation = Task.WhenAny(recommendations)
    cts.Cancel()

    if recommendation then
       do! buyStock symbol
}
```

Task.WhenAny – Interleaving Handle Tasks as they complete

```
let imageTasks urls : Task<Bitmap> list =
    urls |> List.map getBitmapAsync

let imageTasksRes = ResizeArray(imageTasks)

let processImages () = task {

    while imageTasksRes.Count > 0 do
       let! imageTask = Task.WhenAny(imageTasks)

       imageTasks.Remove(imageTask)

       let image = imageTask
       processImage image
}
```

Unbounded parallelism Async.Parallel (and Async.Start)

```
let httpAsync (url : string) = async {
    let req = WebRequest.Create(url)
    let! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    return! reader.ReadToEndAsync() }

let sites =
    [   "http://www.yahoo.com";    "http://www.amazon.com"
        "http://www.google.com";   "http://www.netflix.com";
        "http://www.facebook.com"; "http://www.docs.google.com";
        "http://www.youtube.com";  "http://www.gmail.com";
        "http://www.reddit.com";   "http://www.twitter.com"; ]

|> Seq.map httpAsync
|> Async.Parallel
```

Combinator for-each async

```
let forEachTaskWithResult (source : 'a seq) dop (action : 'a -> Task<'b>) = task {
    let! data =
        Partitioner.Create(source).GetPartitions(dop)
        |> Seq.map(fun partition -> task {
            let localResults = ResizeArray<'b>()
            use p = partition
            while p.MoveNext() do
                let! result = action p.Current
                localResults.Add result
            return localResults
        })
        |> Task.WhenAll

    return data |> Seq.concat |> Seq.toList
    }
```

```
let sendEmailsAsync(emails : string seq) = task {
    let client = new SmtpClient()
    let sendEmailAsync = fun emailTo -> task {
        let message = MailMessage("me@me.com", emailTo)
        do! client.SendMailAsync(message)
       }
    do! forEachTask emails 8 sendEmailAsync
}
```

Working with Async Computation

Async orElse

```
module SerivceCurrencyOne =
        let getRate (ccy : string ) = // ... implementation here

module SerivceCurrencyTwo =
        let getRate (ccy : string ) = // ... implementation here


let orElse (fallBack : exn -> Task<'a>) (op : Task<'a>) = task {
    let! k = op.ContinueWith(fun (t : Task<'a>) ->
        if t.Status = TaskStatus.Faulted then
            fallback t.Exception
        else Task.FromResult(t.Result))
    return k
}

let ccy = "USD"
SerivceCurrencyOne.getRate(ccy) |> Task.orElse (fun _ -> SerivceCurrencyTwo.getRate ccy)
```

Async retry

```
let rec retry (retries :int) (delayMillis : int) (op : unit -> Task<_>) = task {
    match retries with
    | 0 -> return! op()
    | n -> return! op() |> orElse (fun ex ->
                task {
                    do! Task.Delay delayMillis
                    return! retry (n - 1) (delayMillis * 1.5) op
                })
    }
```

```
SerivceCurrencyOne.getRate(ccy) |> Task.retry 10 1500
```

```
SerivceCurrencyOne.getRate(ccy)
|> Task.orElse (fun _ -> SerivceCurrencyOne.getRate ccy)
|> Task.retry 10 1500
```

Async bimap

```fsharp
let bimap (successed : 'a -> 'b) (faulted : exn -> 'b) (op : Task<'a>) = task {
    return! op.ContinueWith(fun (t : Task<'a>) ->
        if t.Status = TaskStatus.Faulted then
            faulted t.Exception
        else successed t.Result)
    }

let bimap' (successed : 'a -> 'b) (faulted : exn -> 'b) (op : Task<'a>) = task {
  return! op
          |> Task.map successed
          |> Task.orElse (fun ex -> faulted ex |> Task.FromResult)
}

let getRate =
    SerivceCurrencyOne.getRate ”AAPL"
    |> Task.bimap (fun rate -> printfn "The rate is %d" rate)
                  (fun ex   -> printfn "Error : %s" ex.Message)
```

All together

```
[<HttpGet("translate/{amount}/{from}/{to}")>]
let translate (amount : decimal, from : string, to : string) = task {
    let ccy = from + to
    return!
        SerivceCurrencyOne.getRate(ccy)
        |> Task.orElse (fun () -> SerivceCurrencyTwo.getRate ccy)
        |> Task.retry 10 1500
        |> Task.map (fun rate -> amount * rate)
        |> Task.bimap (fun result -> StatusCode.Ok result)
                      (fun ex -> StatusCode(500, ex))

}
```

Functional tools for implementing combinators

The functional toolbox

Tool purpose	Tool name	Jargon
Composition	Compose	Monoid
Iteration	Fold	
Aggregation	Combine & Reduce	
Mixing Effects	fmap/map & return	Functor
Chaining Effects	bind/flatMap	Monad
Effects in parallel	apply & zip	Applicative
Pulling Effects out of list	Sequence & Traverse	

Functional Task Toolkit

```fsharp
let retn value = value |> Task.FromResult

let bind (f : 'a -> Task<'b>) (x : Task<'a>) = task {
    let! x = x
    return! f x
}

let apply f x =
  bind (fun f' ->
    bind (fun x' -> retn(f' x')) x) f

let map f x = x |> bind (f >> retn)

let map2 f x y =
  (apply (apply (retn f) x) y)

let map3 f x y z =
  apply (map2 f x y) z


let kleisli f g x = bind (f x) g
```

Monoids

- Associativity, is when combining more than two things, which pairwise combination you do first doesn't matter.
  - Benefit of Associativity: Divide and Conquer, parallelization, and incremental accumulation

1 + 2 = 3

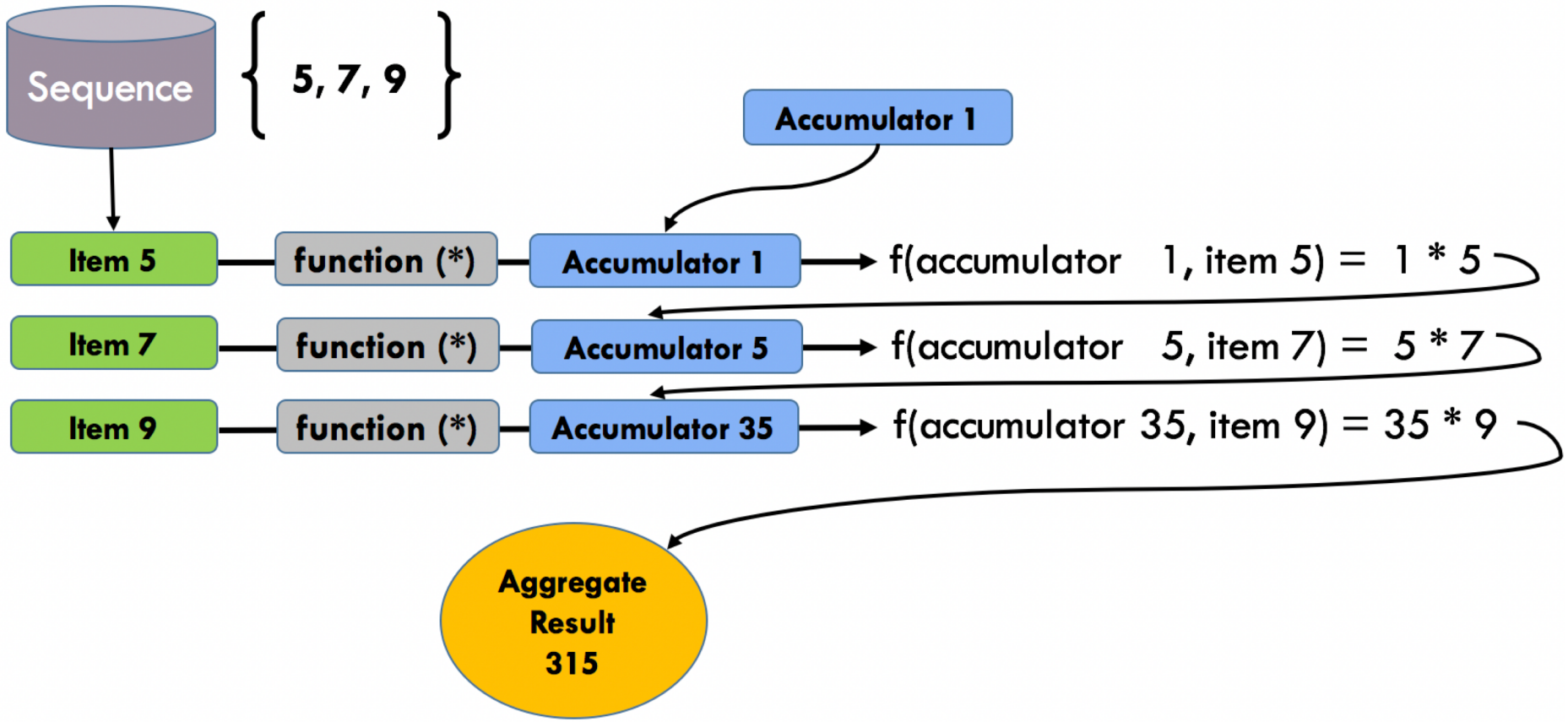
1 + (2 + 3) = (1 + 2) + 3

- Identity is a special thing called "Zero" such that when you combine any thing with "Zero" you get the original thing back.

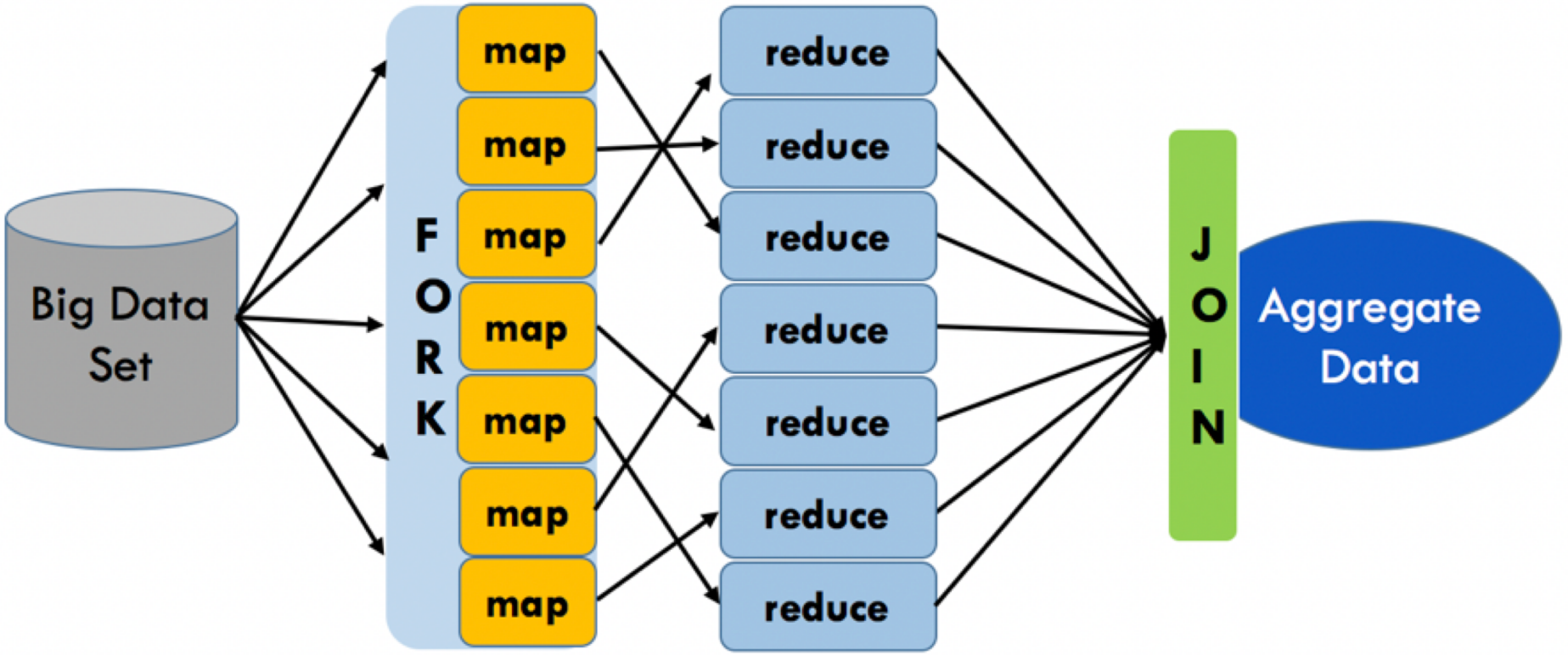
1 + 0 = 1 and 0 + 1 = 1

```
module String =
    let mempty = "" // zero
    let mappend x y = sprintf "%s%s" x y // combine
    // aggregate function implicit first element as initial seed.
    let ofList = List.fold mappend mempty

// the list does not provide a function with implicit zero as head,
// The monoid is implicit but necessary when you fold/traverse structure
String.mappend "hello" " world" = (String.ofList ["hello"; " "; "world"])
```

Aggregating and reducing



Map Reduce


Big Data Set
FORK
map
map
map
map
map
map
reduce
reduce
reduce
reduce
reduce
reduce
JOIN
Aggregate Data

Implementing Parallel Map Reduce

```
// mapReduce : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a [] -> 'b
let mapReduce f g xs =
    Array.map f xs  |> Array.reduce g

let reduce f xs = mapReduce id f xs
let minBy f xs = mapReduce f min xs
let maxBy f xs = mapReduce f max xs
```

```
let mapReducePar f g xs =
    Array.Parallel.map f xs
    |> Array.reduce g
```

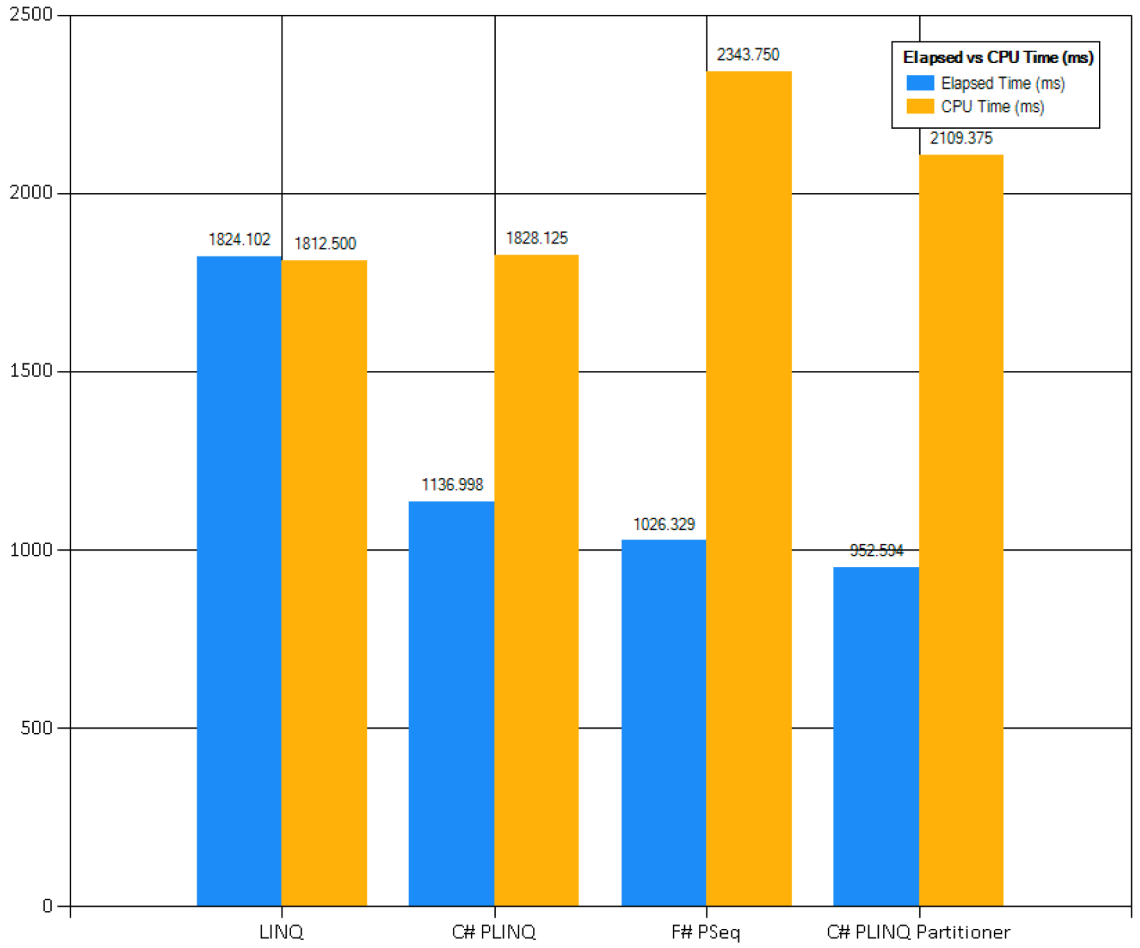
Implementing Parallel Reduce

```
// ('a -> 'b -> 'c) -> Task<'a> -> Task<'b> -> Task<'c>
let map2 f x y =
  (apply (apply (retn f) x) y)

let parallelReduce depth f (ie :'a array) = = task {

  let rec reduceRec depth f (ie :'a array) = function
    | 1 -> ie.[0]
    | 2 -> f ie.[0] ie.[1]
    | len ->
      let h = len / 2
      if depth < 0 then
          let o1 = reduceRec (depth - 1) f (ie |> Array.take h) h)
          let o2 = reduceRec (depth - 1) f (ie |> Array.skip h) (len-h))
          f o1 o2
      else
          let o1 = Task.Run(fun _ -> reduceRec f (ie |> Array.take h) h)
          let o2 = Task.Run(fun _ -> reduceRec f (ie |> Array.skip h) (len-h))
          return! Task.map2 f o1 o2
   return! reduceRec depth f ie ie.length
}
```

Map-Reduce in F#

```
let mapF  M (map:'in_value -> seq<'out_key * 'out_value>)  (inputs:seq<'in_value>) =
    inputs
    |> PSeq.withExecutionMode ParallelExecutionMode.ForceParallelism
    |> PSeq.withDegreeOfParallelism M
    |> PSeq.collect (map)
    |> PSeq.groupBy (fst)
    |> PSeq.toList


let mapReduce
        (inputs:seq<'in_value>)
        (map:'in_value -> seq<'out_key * 'out_value>)
        (reduce:'out_key -> seq<'out_value> -> 'reducedValues)
        M R =

    inputs |> (mapF M map >> parallelReduce R reduce)
```

Map-Reduce in action



Functor

- Crates a new “enriched” type from any type
- Lifting of functions
- For any function (a -> b) creates a function acting on “enriched” types
- Using Functor you don’t need to break encapsulation

('a ➔ 'b) ➔ Async<'a> ➔ Async<‘b>

('T ➔ 'R) ➔ Task<T> ➔ Task<R>

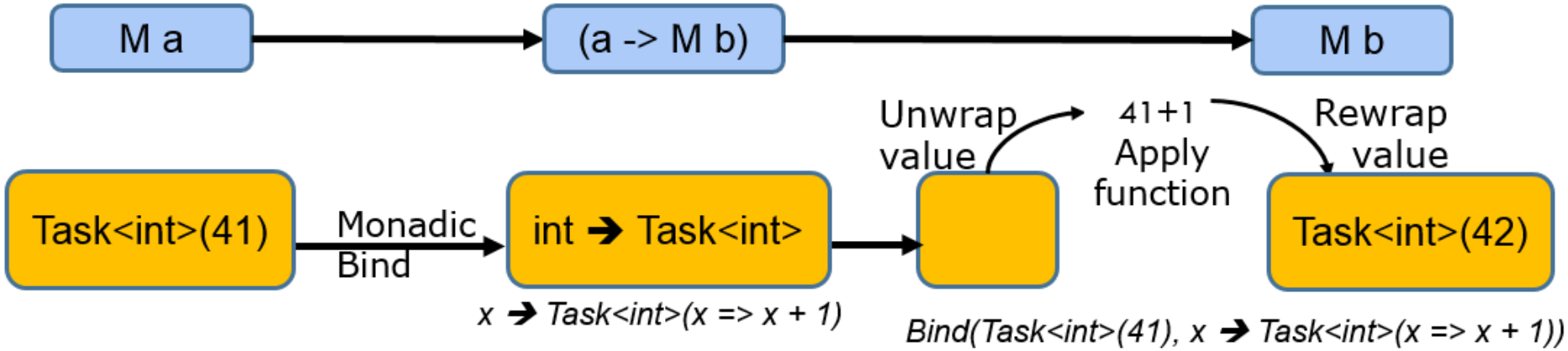
```
module Async =
    let map (func:'a -> 'b) (operation:Async<'a>) : Async<‘b> = async {
        let! result = operation
        return func result }

let transformImage (blobNameSource:string) =
    downloadBitmapAsync(blobNameSource)
    |> Async.map ImageHelpers.setGrayscale
    |> Async.map ImageHelpers.createThumbnail
```

Monad

is a generic type

- With function (>>=): bind : M<‘a> -> (‘a -> M<‘b>) -> M<‘b>
- With function: return : ‘a -> M<‘a>

```
Bind (f : ‘a -> Task<‘b>) (Task<‘a> m) : Task<‘b>
Return (‘a value) : Task<‘a>
```



Monad

```
let fetchLines (url: string) i = async {
    let rec append i (s: StreamReader) (b: StringBuilder) =
        match i with
        | 0 -> b.ToString() | _ -> s.ReadLine()
        |> b.AppendLine
        |> append (i-1) s
    let req = WebRequest.Create(url)
    use! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    return append i reader (StringBuilder())}
```

```
async.Bind(fetchLines "http://microsoft.github.io" 10, fun x ->
    async.Bind(fetchLines "http://fsharp.org" 10, fun y ->
        async.Bind(fetchLines "http://funscript.info" 10, fun z ->
            async.Return(x + y + z))))
                |> Async.RunSynchronously
```

Computation Expression (Sugaring CPS)

In F# the generic monadic type is replaced and defined with the computation expression, in the case of Async, CE is defined with the specialized type Async:

Method	Typical signature	Description
Bind	M<'T> * ('T -> M<'U>) -> M<'U>	Called for let! and do! in computation expressions.
Return	'T -> M<'T>	Called for return in computation expressions.

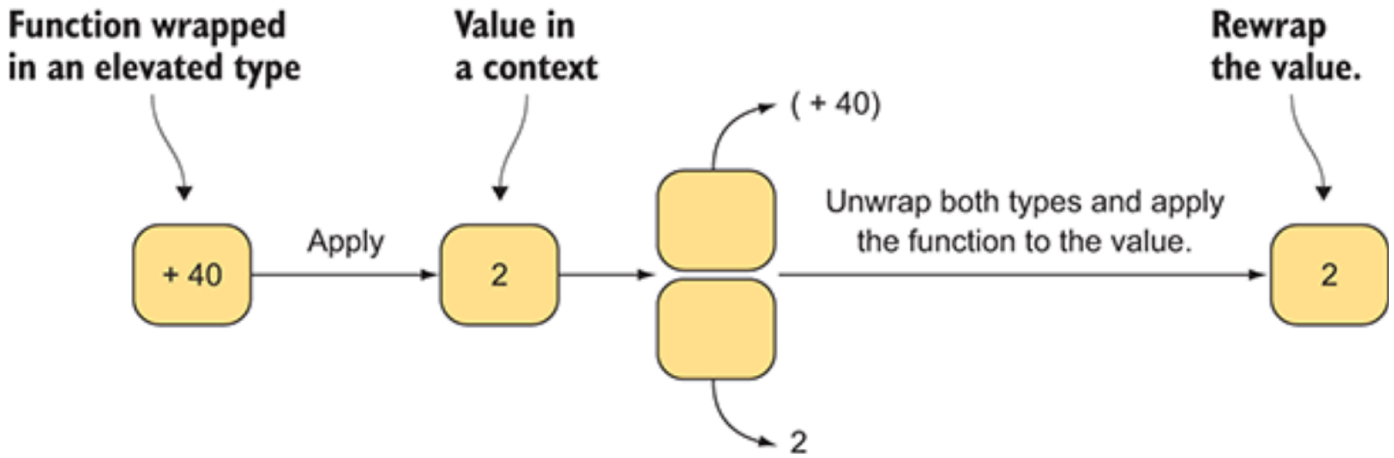
```
let bind (f : 'a -> Task<'b>) (x : Task<'a>) = task {
    let! x = x
    return! f x
}

let retn (x : 'a) = Task.FromResult x
```

Computation Expression

```
let fetchLines (url: string) i = async {
    let rec append i (s: StreamReader) (b: StringBuilder) =
        match i with
        | 0 -> b.ToString() | _ -> s.ReadLine()
        |> b.AppendLine
        |> append (i-1) s
    let req = WebRequest.Create(url)
    use! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    return append i reader (StringBuilder())}


// In F# the boiler plate code has already been done for us in Async
// But we can implement our own custom computation expressions easily
async { let! x = fetchLines "http://microsoft.github.io" 10
        let! y = fetchLines "http://fsharp.org" 10
        let url = "http://funscript.info"
        let! z = fetchLines url 10
        return x + y + z }
|> Async.RunSynchronously
```

Applicative

```
val pure  : 'T -> F<'T>
val apply : AF<T -> R> -> AF<T> -> F<R>
val (<*>) : apply
```



The Apply operator implements the function wrapped inside an elevated type to a value in the context. The process triggers the unwrapping of both values; then, because the first value is a function, it's applied automatically to the second value. Finally, the output is wrapped back inside the context of the elevated type.

Applicative

```
let retn x = async { return x }


I.  // Async<'a -> 'b> -> Async<'a> -> Async<'b>
    let apply af ax = async {
        // We start both async task in Parallel
        let! pf = Async.StartChild af
        let! px = Async.StartChild ax
        // We then wait that both async operations complete
        let! f = pf
        let! x = px
        // Finally we execute (f x)
        return f x
    }

    let (<*>) = apply
```

Applicative

```
let blendImages (blobReferenceOne:string) (blobReferenceTwo:string) (size:Size) =
        Async.apply(
            Async.apply(
                Async.apply(
                    Async.pure blendImages)
                                (downloadOptionImage(blobReferenceOne)))
                                (downloadOptionImage(blobReferenceTwo)))
                (Async.pure size)
```

```
let blendImages (blobReferenceOne:string) (blobReferenceTwo:string) (size:Size) =
     blendImages
     <!> downloadOptionImage(blobReferenceOne)
     <*> downloadOptionImage(blobReferenceOne)
     <*> Async.``pure`` size
```

Heterogenous Parallel computations (lift_)

```
let lift2 f x y = f <!> x <*> y

let lift3 (func:'a -> 'b -> 'c -> 'd) (asyncA:Async<'a>)
          (asyncB:Async<'b>) (asyncC:Async<'c>) =
    func <!> asyncA <*> asyncB <*> asyncC
```

```
Async.lift3 (fun x y z -> String.length x + String.length y + String.length z)
            fetchLines "http://microsoft.github.io" 10
            fetchLines "http://fsharp.org" 10
            fetchLines "http://funscript.info" 10
```

Running asynchronous operations in parallel with applicative and monad

Parallel async operations with applicative

```
type Flight = { airline : string; price : decimal }

type Airline =
    abstract fligths : string -> string -> DateTime -> Task<Flight seq>
    abstract bestFare : string -> string -> DateTime -> Task<Flight>
```

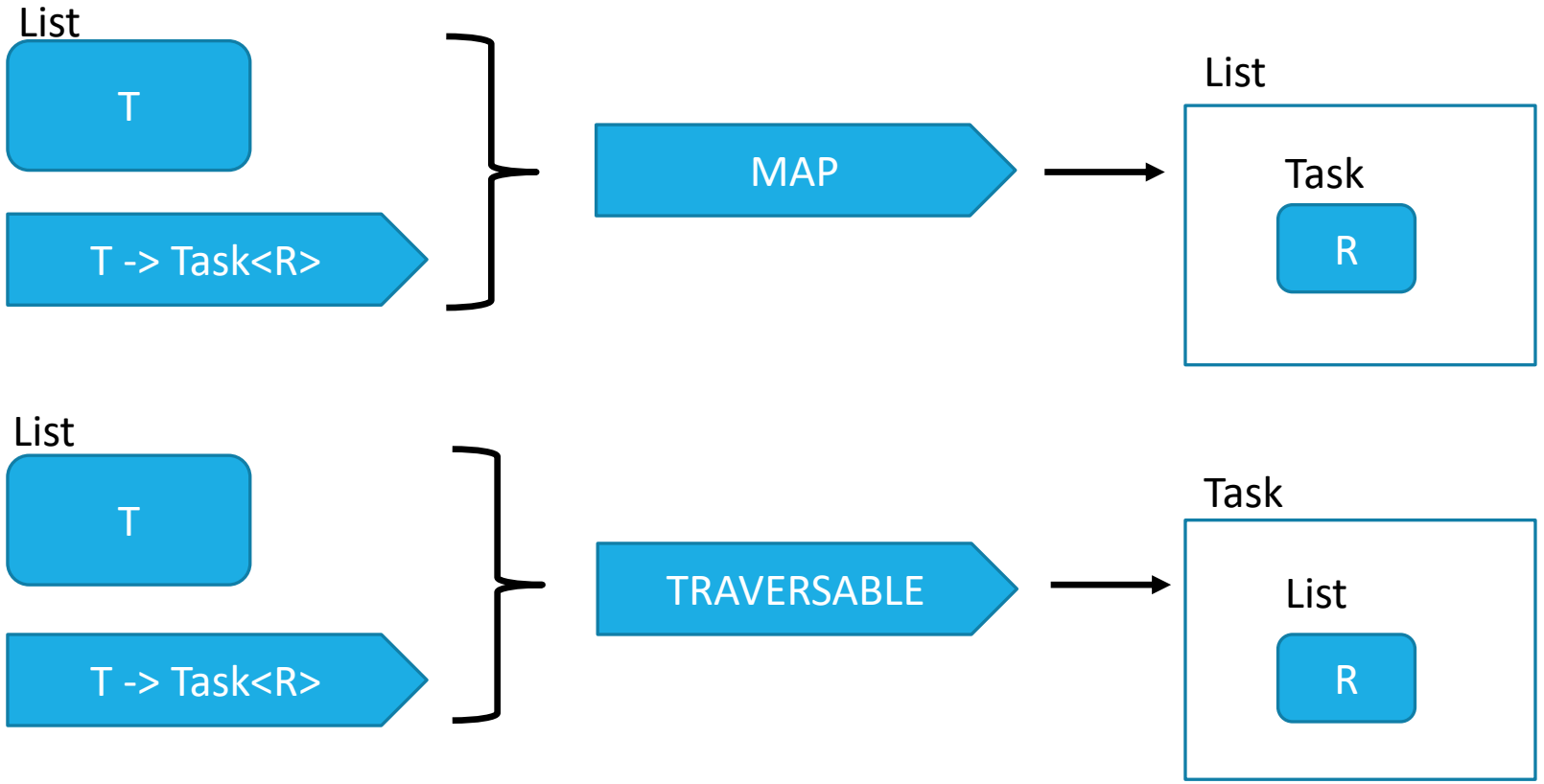
Parallel async operations with monad

```
type Flight = { airline : string; price : decimal }

type Airline =
    abstract fligths : string -> string -> DateTime -> Task<Flight seq>
    abstract bestFare : string -> string -> DateTime -> Task<Flight>



let delta : Airline = // ... implementation here
let americans : Airline = // ... implementation here

let bestFareM (from : string) (to' : string) (on : DateTime) = task {
    let! d = delta.bestFare from to' on
    let! a = americans.bestFare from to' on
    return
        if d.price > a.price then a else d

}
```

Parallel async operations with applicative

```
let retn value = value |> Task.FromResult

let apply af ax = async {
    let! pf = Async.StartChild af
    let! px = Async.StartChild ax
    let! f = pf
    let! x = px
    return f x
}



let pickCheaper a b =
    if a.price > b.price then b else a

let bestFareA (from : string) (to' : string) (on : DateTime) = task {
    return! Task.retn pickCheaper
    |> Task.apply (delta.bestFare from to' on)
    |> Task.apply (americans.bestFare from to' on)
}
```

Traversable parallel async operations



Traversable parallel async operations

```
let retn x = async { return x }

let apply af ax = async {
    let! pf = Async.StartChild af
    let! px = Async.StartChild ax
    let! f = pf
    let! x = px
    return f x
}

let (<*>) = apply

let traverse f list =
    let folder x xs = retn (fun x xs -> x :: xs) <*> f x <*> xs
    List.foldBack folder list (retn [])
```

Traversable parallel async operations

```
Tr<T> → (T → A<R>) → A<Tr<R>>
List<T> → (T → Task<R>) → Task<List<R>>
```

```
let rec traverseTasktM f list =
    // define the monadic functions
    let (>>=) x f = bind f x

    // define a "cons" function
    let cons head tail = head :: tail
    // loop through the list
    match list with
    | [] -> retn []
    | head::tail ->
        // lift the head using f
        // then lift the tail using traverse
        // then cons the head and tail and return it
        f head >>= (fun h ->
            traverseTasktM f tail >>= (fun t ->
                retn (cons h t) ))
```

Traversable parallel async operations

```
Tr<T> → (T → A<R>) → A<Tr<R>>
List<T> → (T → Task<R>) → Task<List<R>>
```

```
let rec traverseTaskA f list =
    // define the applicative functions
    let (<*>) = apply


    // define a "cons" function
    let cons head tail = head :: tail

    // loop through the list
    match list with
    | [] -> retn []
    | head::tail ->
        // combine the effects using applicative
        retn cons <*> (f head) <*> (traverseTaskA f tail)
```

Traversable parallel async operations

```
let airlines = Unchecked.defaultof<Airline seq>

let map f x = x |> bind (f >> singleton)

let flights from to' on = airlines |> Seq.map (fun a -> a.fligths from to' on)



let search (airlines : Airline list) from to' on = task {
    let! flights = traverseTaskA airlines (fun a ->
        a.flights from to' on
        |> Async.orElse (fun ex -> Seq.empty))
    return flights |> Seq.collect id |> Seq.sortByDescending (fun p -> p.price)
}
```

Applicative vs Monad

Monad (bind) : the first Task must be awaited in order to create the second task, so it should be used when the creation of a Task depends on the return value another.

Applicative : both tasks are provided by the caller, so you should use it when the tasks can be started independently.

Kliesli operator in F#

```
let (>>)  f g x = (f x) |> g   // This is how normal composition is defined
let (>=>) f g x = (f x) >>= g  // This is Kleisli composition
let kleisli f g x = bind (f x) g
```

```
Bind : ('T -> Async<'R>) -> Async<'T> -> Async<'R>
Bind : ('T -> Task<‘R>)  -> Task<‘T>  -> Task<'R>
```

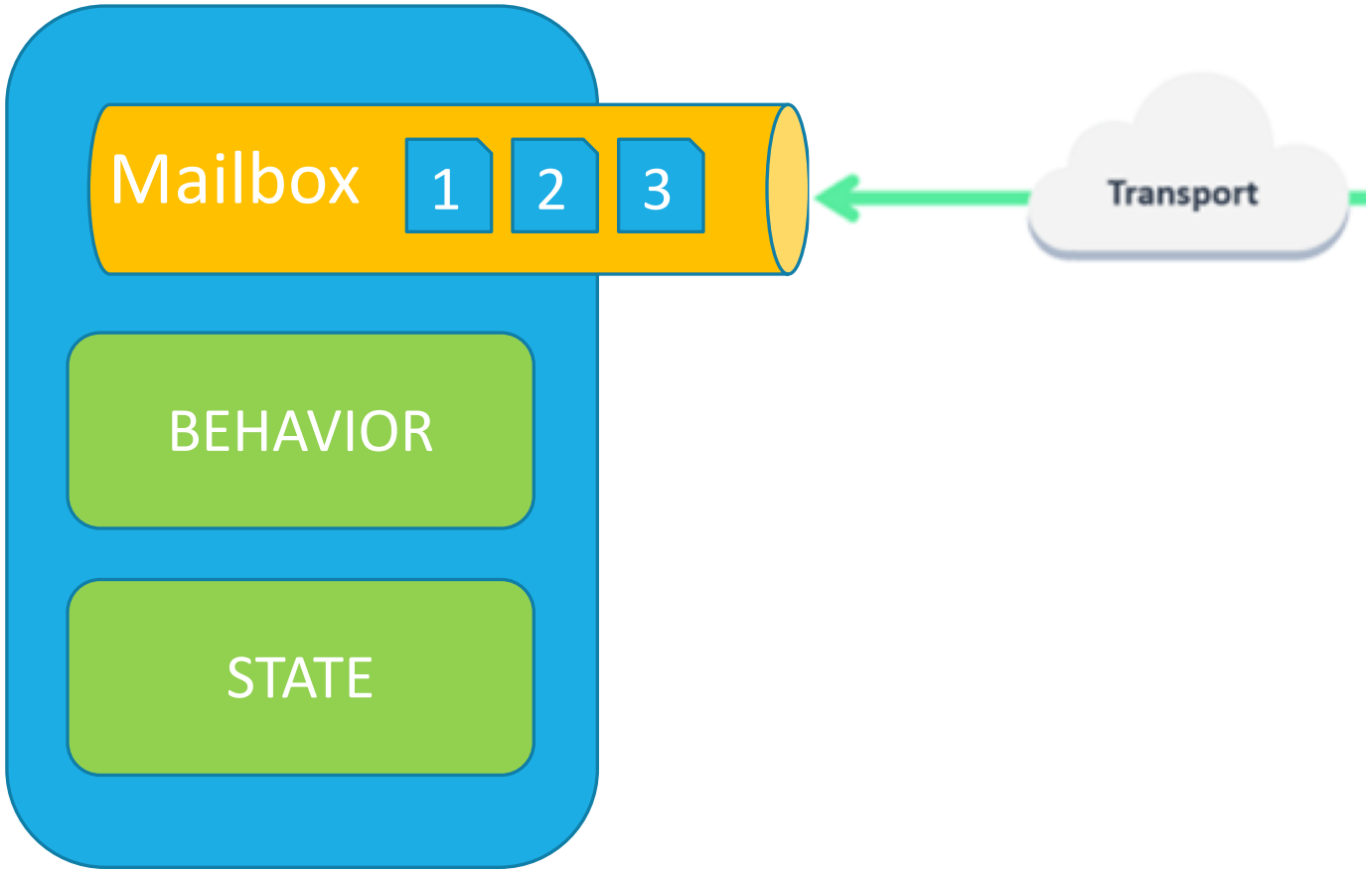
```
Kleisli (>=>) : ('T -> Async<'R>) -> ('TR -> Async<‘RR>) -> ('T -> Async<‘RR>)
Kleisli (>=>) : ('T -> Task<‘R>)  -> ('TR -> Task<'RR>)  -> ('T -> Task<‘RR>)

('T -> Async<'R>) >=> ('TR -> Async<‘RR>)
('T -> Task<‘R>)  >=> ('TR -> Task<‘RR>)
```

Kliesli operator in F#

```
fAsync:('a -> Async<'b>) -> gAsync:('b -> Async<'c>) -> arg:'a -> Async<‘c>

let Kliesli (fAsync:'a -> Async<'b>) (gAsync:'b -> Async<'c>) (arg:'a) =
    async {
        let! f = Async.StartChild (fAsync arg)
        let! result = f
        return! gAsync result }

let (>=>) = Kliesli



let processStockHistory =
       downloadStockHistoryAsync >=> convertStockHistory
```

Agent Patterns and composition

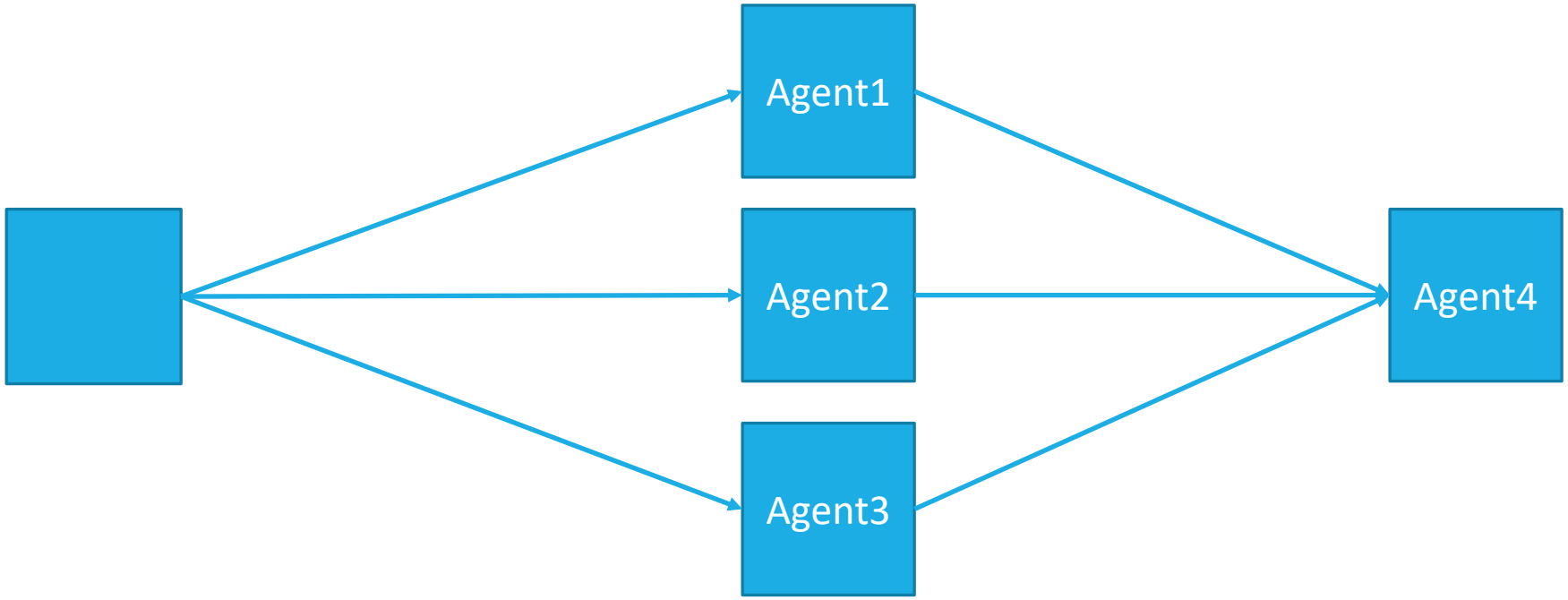
Message Passing based concurrency



- Processing
- Storage – State
- Communication only by messages
- Share Nothing
- Message are passed by value
- Lightweight object
- Running on it’s own thread
- No shared state
- Messages are kept in mailbox and processed in order
- Massively scalable and lightening fast because of the small call stack

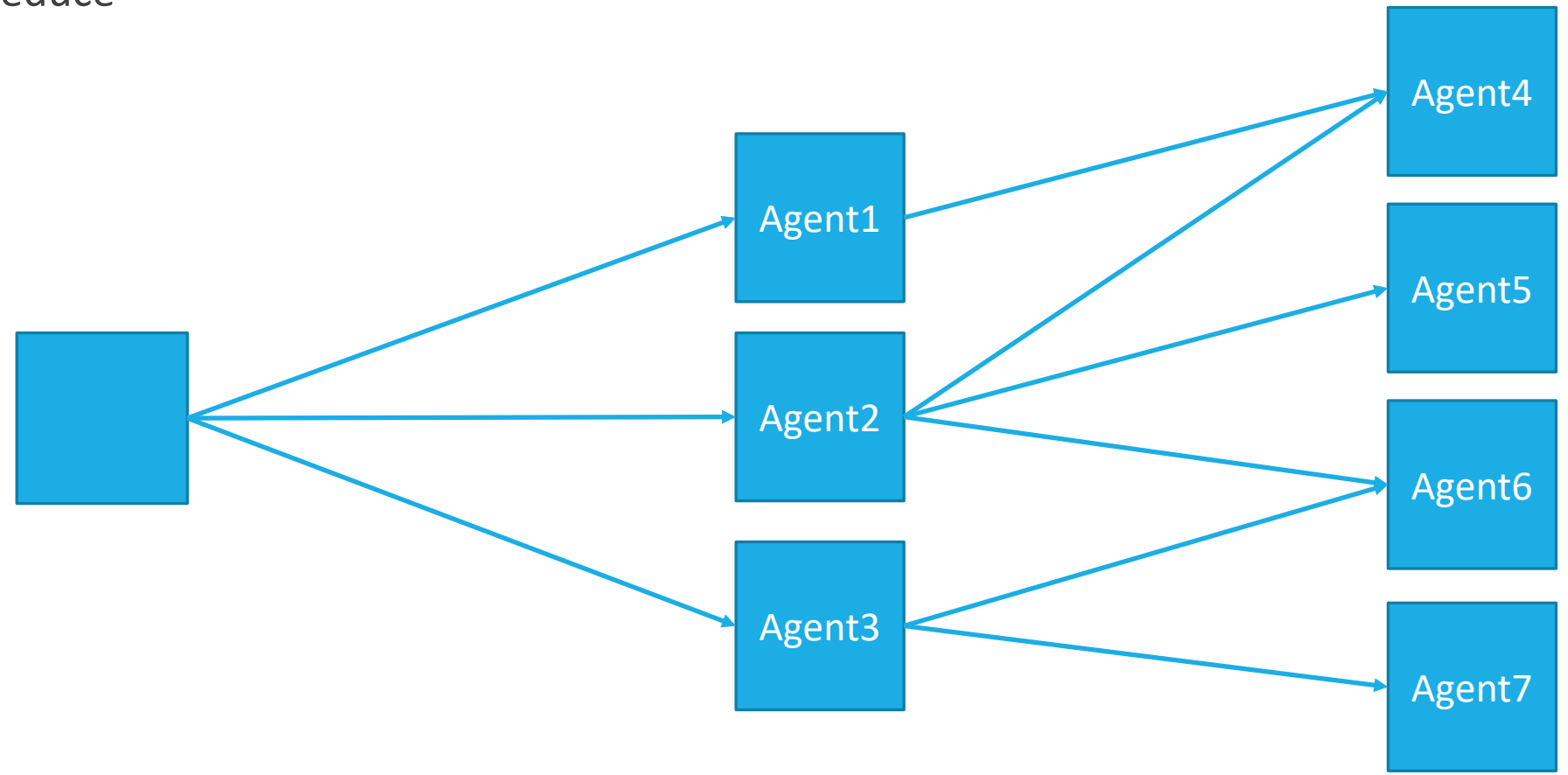
Agent patterns

Fork Join



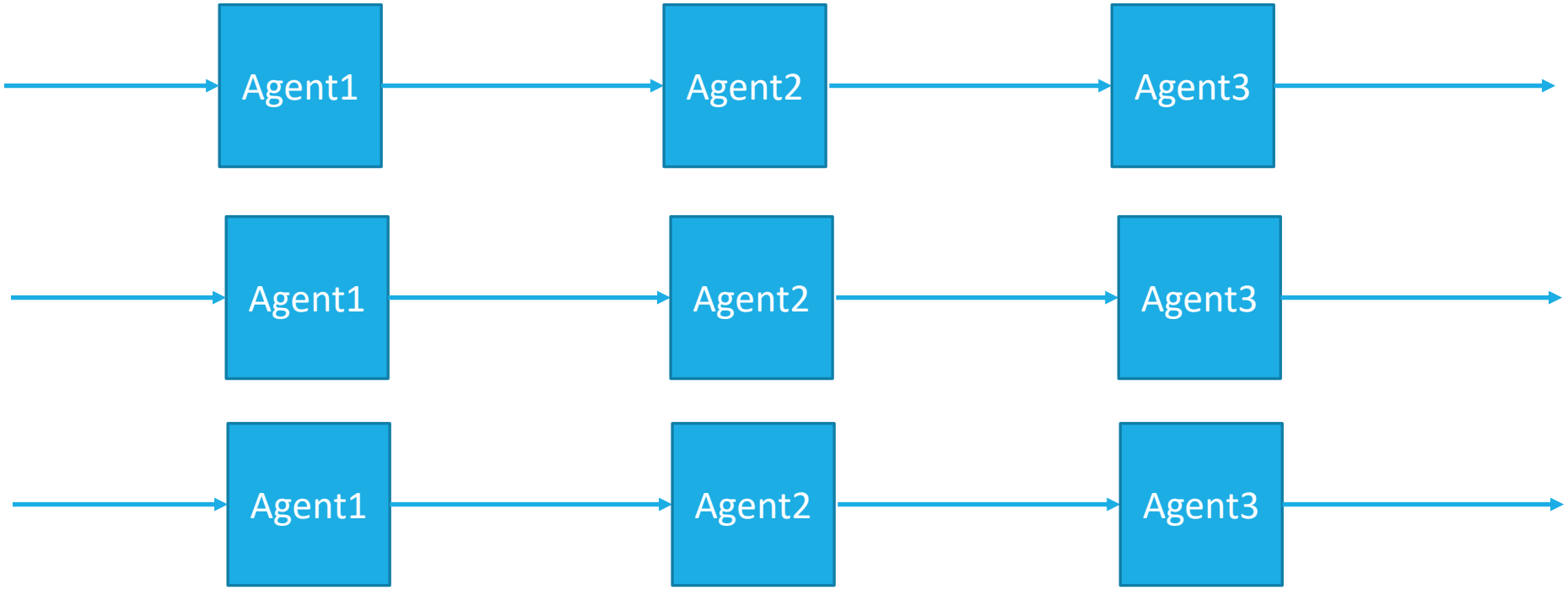
Agent patterns

Map Reduce

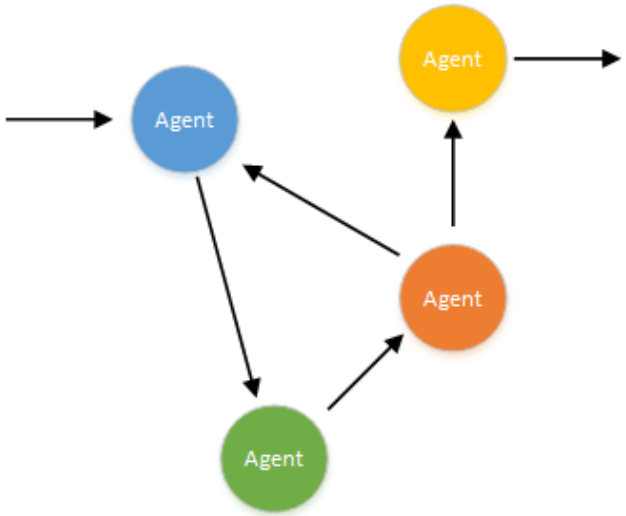


Agent patterns

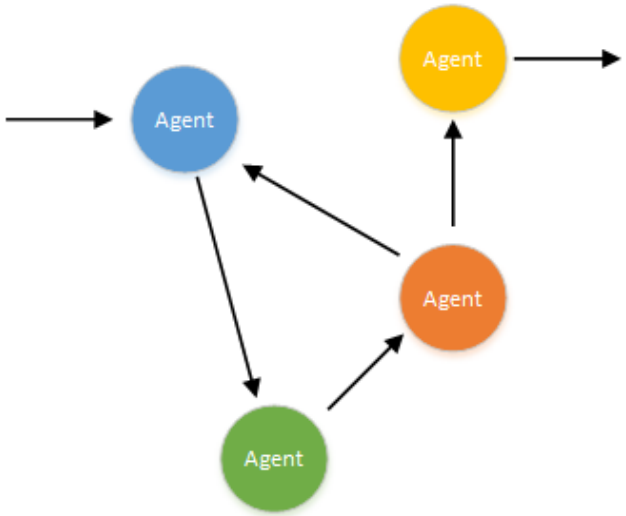
Pipeline



Agent Pipeline



Agent Pipeline



Agent Pipeline

```
let agent f = Agent<Message>.Start(fun inbox ->
    let rec loop () = async {
        let! msg, replyChannel = inbox.Receive()

        f msg
        replyChannel.Reply msg

        return! loop() }
    loop() )
```

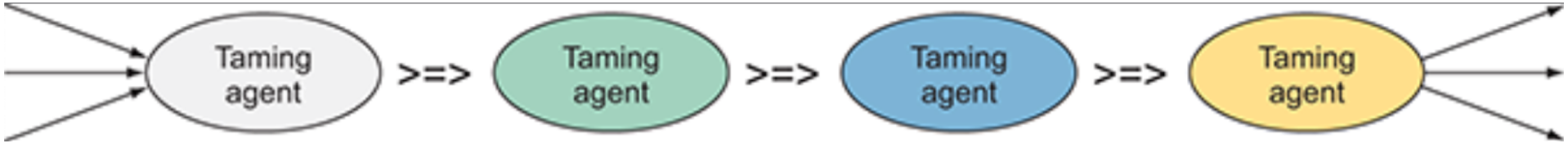
Agent Pipeline

```
let agent f = Agent<Message>.Start(fun inbox ->
    let rec loop () = async {
        let! msg, replyChannel = inbox.Receive()
        f msg
        replyChannel.Reply msg
        return! loop() }
    loop() )


let pipelineAgent (operation:'a -> Async<'b>) : ('a -> Async<'b>) =
    let agent = agent f
    fun (job: 'a) -> agent.PostAndAsyncReply(fun replyChannel ->  job, replyChannel)

let agent1 = pipelineAgent (printfn "Pipeline processing 1: %s")
let agent2 = pipelineAgent (printfn "Pipeline processing 2: %s")

let message = "Message in the pipeline"

let pipeline x = Async.retn x >>= agent1 >>= agent2
```

Agent Binding

```
let (>>=) x f = Async.bind f x
let pipeline x = Async.retn x >>= agent1 >>= agent2
```

```
let kleisli f g x = Async.bind (f x) g
let (>=>) f1 f2 x = f1 x >>= f2
let pipeline = agent1 >=> agent2

pipeline message
```




f10
delete
DEMO

Controlling execution flow with concurrent combinators

Controlling flow with conditional concurrent combinators

You could implement an asynchronous combinator that emulates an **if-else** statement equivalent to the imperative conditional logic, but how?

The solution is found in the functional patterns:

- Monoids can be used to create the **OR** combinator.
- Applicative can be used to create the **And** combinator.
- Monads chain asynchronous operations and glue combinatory.

The async AND combinator

Returns result after both functions finish (like wait for all)

- Call the first one, wait for result, then call the second one, combine results
- Unless the functions have side effects, result is deterministic and independent of order
- **ANDs** can be combined into chains as long as they all return the same types: result = a & b & c
- Can be generalized to multiple types

The async OR combinator

Return the first result that arrives (like wait for any or select)

- Start the two calls in parallel. When one of them returns, return its result
- It has to use threads, but what about the second thread termination
- Nondeterministic, unless both are guaranteed to return the same value
- **ORs** can be chained: result = a | b | C

Controlling flow with conditional concurrent combinators

```
let inline ifAsync (predicate:Async<bool>) (funcA:Async<'a>)  (funcB:Async<’a>) =
       async.Bind(predicate, fun p -> if p then funcA else funcB)

let inline iffAsync (predicate:Async<'a -> bool>) (context:Async<'a>) = async {
       let! p = predicate <*> context
       return if p then Some context else None }

let inline notAsync (predicate:Async<bool>) = async.Bind(predicate, not >> async.Return)
```

Controlling flow with conditional concurrent combinators

```
let inline ifAsync (predicate:Async<bool>) (funcA:Async<'a>)  (funcB:Async<’a>) =
       async.Bind(predicate, fun p -> if p then funcA else funcB)

let inline iffAsync (predicate:Async<'a -> bool>) (context:Async<'a>) = async {
       let! p = predicate <*> context
       return if p then Some context else None }

let inline notAsync (predicate:Async<bool>) = async.Bind(predicate, not >> async.Return)

let inline AND (funcA:Async<‘a>) (funcB:Async<‘a>) =
           ifAsync (async.Return false) funcA funcB
let (<&&>)(funcA:Async<‘a>) (funcB:Async<‘a>) = AND funcA funcB
```

Controlling flow with conditional concurrent combinators

```
let inline ifAsync (predicate:Async<bool>) (funcA:Async<'a>)  (funcB:Async<'a>) =
       async.Bind(predicate, fun p -> if p then funcA else funcB)

let inline iffAsync (predicate:Async<'a -> bool>) (context:Async<'a>) = async {
       let! p = predicate <*> context
       return if p then Some context else None }

let inline notAsync (predicate:Async<bool>) = async.Bind(predicate, not >> async.Return)

let inline AND (funcA:Async<'a>) (funcB:Async<'b>) =
            ifAsync funcA funcB (async.Return false)
let (<&&>)(funcA:Async<'a>) (funcB:Async<'b>) = AND funcA funcB

let inline OR (funcA:Async<'a>) (funcB:Async<'a>) =
   ifAsync funcA (async.Return true) funcB
let (<||>)(funcA:Async<'a>) (funcB:Async<'a>) = OR funcA funcB
```

Use case
Stock market treading

Async combinators in action
Stock trade decision

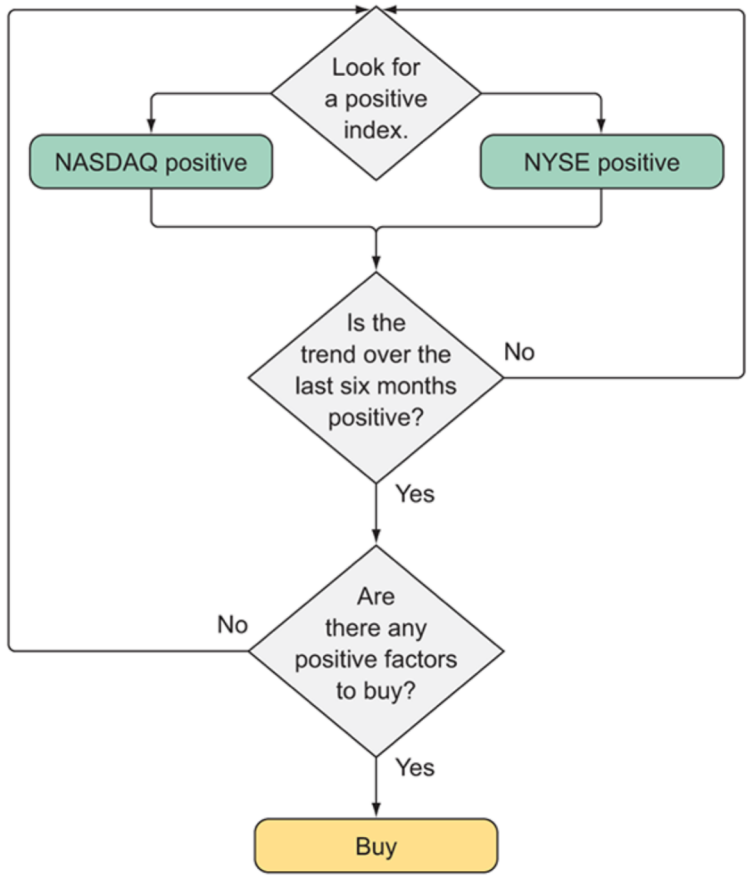
Check the total amount available for purchase based on the bank account available balance and the current price of the stock:

- Fetch the bank account balance.
- Fetch the stock price from the stock market.

Validate if a given stock symbol is recommended to buy:

- Analyze the market indexes.
- Analyze the historical trend of the given stock.

Given a stock ticker symbol, decide to buy or not buy a certain number of stock options based upon the money available calculated in step 1.



Async combinators in action
Stock trade decision

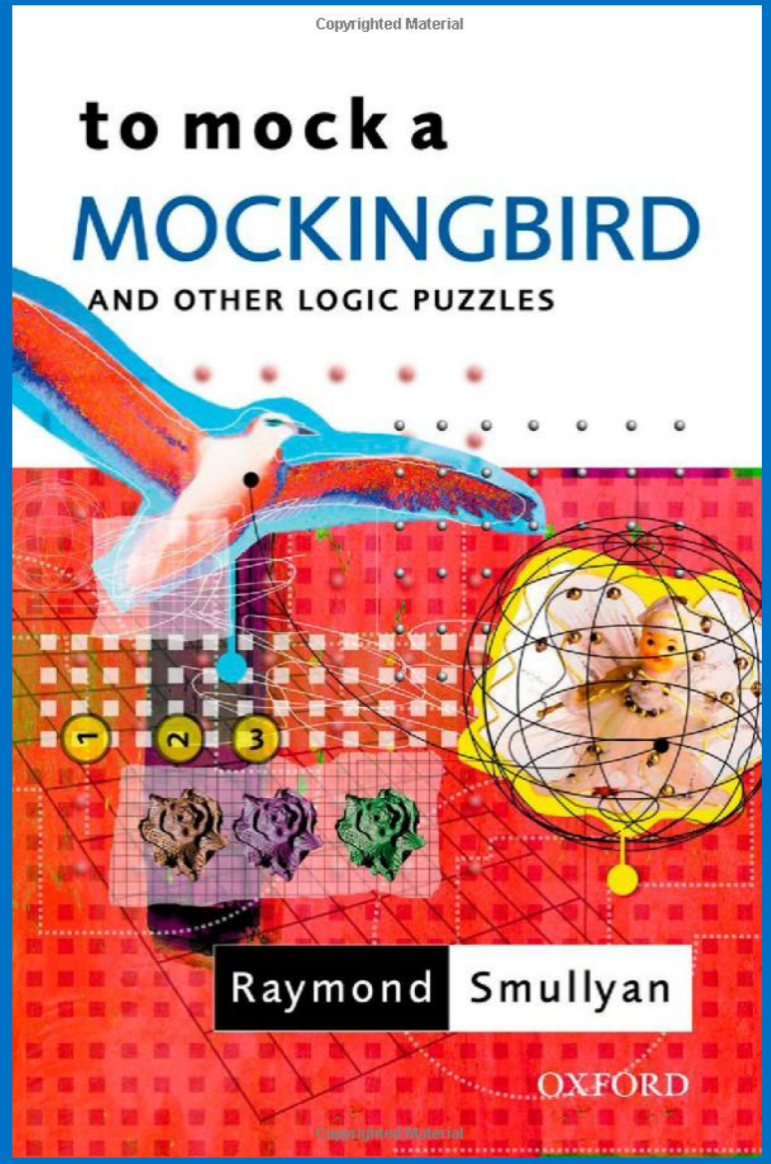
```
let doInvest stockId =
    let shouldIBuy =
        (   (getStockIndex "^IXIC" |> gt 6200.0)
            <|||>
            (getStockIndex "^NYA" |> gt 11700.0 )
        )
        <&&&> ((analyzeHistoricalTrend stockId) |> gt 10.0)
              |> AsyncResult.defaultValue false

    let buy amount = async {
            let! price = getCurrentPrice stockId
            let! result = withdraw (price*float(amount))
            return result |> Result.bimap (fun x -> if x then amount else 0)

                                          (fun _ -> 0)
        }
    Async.ifAsync shouldIBuy
        (buy <!> (howMuchToBuy stockId))
        (Async.retn <| Error(Exception("Do not do it now")))
    |> AsyncResult.handler
```


f10
delete
DEMO

Summary

- We implement a series of combinators to simplify concurrent programming model
- We encapsulate Asynchronous operations
- We compose async/task operations with little effort
- We implement few conditional asynchronous combinators

Combinators Birds



combinators… combinators… combinators

```
/// Identity function, or the Idiot bird
A I<A>(A x) => x

/// The Kestrel
Func<B, A> K<A, B>(A x) => (B y) => x

/// The Mockingbird
Func<A, A> M<A>(Func<A, A> x) => a => x(x(a))

/// The Thrush
Func<Func<A, B>, B> T<A, B>(A x) => (Func<A, B> y) => y(x)

/// The Queer bird
Func<Func<B, C>, Func<A, C>> Q<A, B, C>(Func<A, B> x) => (Func<B, C> y) => (A z) => y(x(z))

/// The Starling
Func<Func<A, B>, Func<A, C>> S<A, B, C>(Func<A, B, C> x) => (Func<A, B> y) => (A z) => x(z, y(z))

/// The infamous Y-combinator, or Sage bird
Func<A, B> Y<A, B, C>(Func<Func<A, B>, A, B> f) => (A x) => f(Y<A, B, C>(f), x)
```

contacts

Source	https://github.com/rikace/concombinators
Twitter	*@trikace*
Blog	*www.rickyterrell.com*
Email	*tericcardo@gmail.com*
Github	*www.github.com/rikace/*

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra