# Monads – Part I

Radu Nicolescu
Department of Computer Science
University of Auckland

9 April 2019//10 April 2019

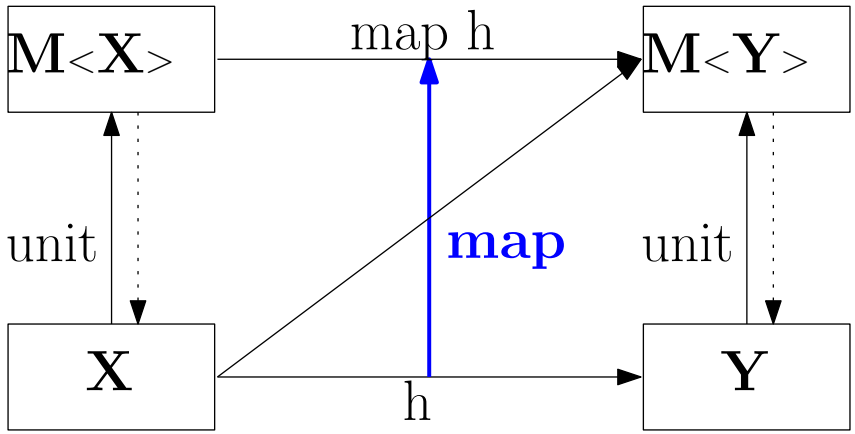
# Current State = Chaos ? Order !

- Base types: int, string, Order, ...
- Functions on base types, e.g. t $\rightarrow$ t+1
- Larger related types: lists, arrays, tasks, asyncs, actors, ...
- Related functions on these types: (map)
  [3;7] $\rightarrow$ [4;8], **async**{**return** 3} $\rightarrow$ **async**{**return** 4}
- Other functions on these types: (flat)
  [[3;4]; [7;8]] $\rightarrow$ [3;4;7;8],
  **async**{**return async**{**return** 3}} $\rightarrow$ **async**{**return** 3}
- Functions that cross boundaries: (unit, unit$^{-1}$)
  3 $\rightarrow$ [3] $\rightarrow$ 3, 3 $\rightarrow$ **async**{**return** 3} $\rightarrow$ 3
- Workflows... **let**! **do**! **return return**! ...
- A bit of systematic order? Monads come to the rescue!

# Monads – Informal Introduction

- Monads are a generic type construction mechanism
- Based on category theory (the mathematics of mathematics)
- Specfic to functional programming
- Formalise the concept of natural transformation
- Orthogonal (complementary) to object-oriented programming
- We use M<X> to represent a generic monad such as
  list <**int**>, list <**string**>, **async**<**int**>, **async**<**string**> ...
- In F# programming: X = 'T = ^T = **int** = ... as needed

# Monads – Unit



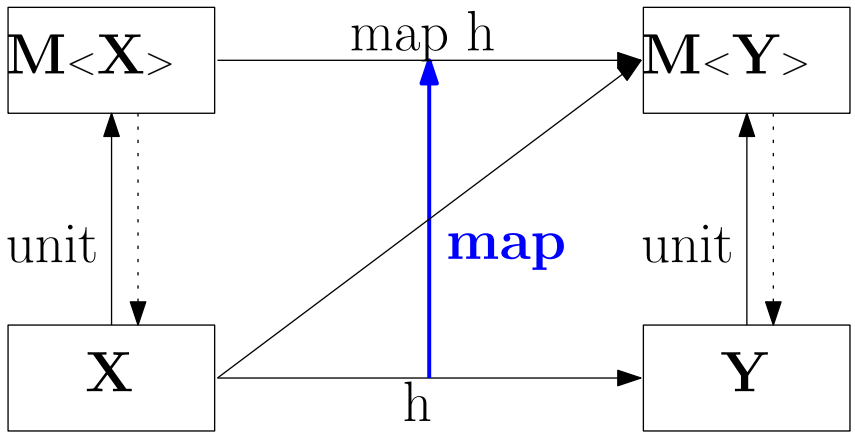

- Each monad M<X> has a "natural" unit; e.g., for M = list

```
1 unit : 'T -> list<'T>  // X -> Y, here X = Y = int
2
3 let inline unit t = [t]  // 3 → [3]
```

- Usually, also a partial inverse, $\mathbf{unit}^{-1}$ (dotted)

```
1 unit⁻¹ [ t ] = t // [3] → 3
2
3 unit⁻¹ [], unit⁻¹ [t1, t2] = undefined
```

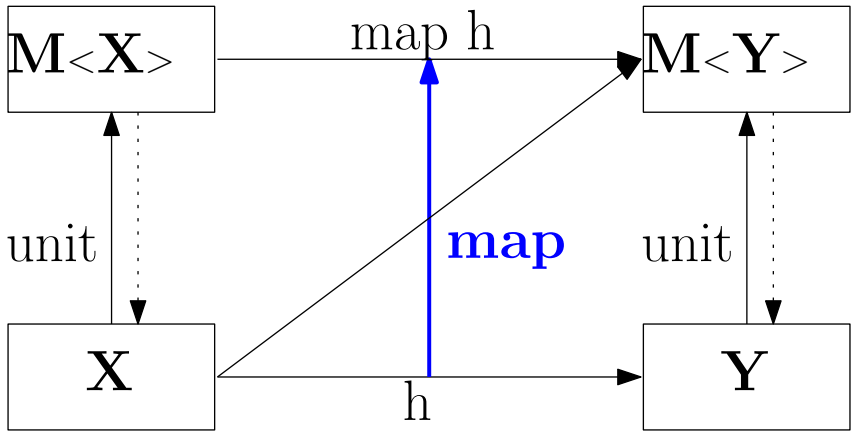
# Monads – High-Order Map




- Each monad M<X> has a "natural" high-order map

```
1  let h t = t + 1   // 3 → 4, h: int -> int
2
3  List.map h : list<int> -> list<int>
4  // [3;7] → [4;8]
5
6  List.map : ('T -> 'U) -> (list <'T> -> list <'U>)
```

# Monads – Diagram Chasing




- Diagram chasing — find equivalent commutative paths
- Our diagrams will be commutative (unless otherwise stated)

```
1  unit >> map h ≡ h >> unit
```

- Proofs

```
1  t  --unit-->  [t]  --map h-->  [t+1]
```

```
1  t  --h-->  t+1  --unit-->  [t+1]
```

# Inline Functions – statically resolved generic types

- Some of our examples use **inline** functions
- Faster... and offer statically resolved generic types
- Standard generic type parameter `'T`
- Statically resolved generic type parameter `^T`

# Standard generic type parameters

- Standard generic type parameter `'T` (apostrophe)

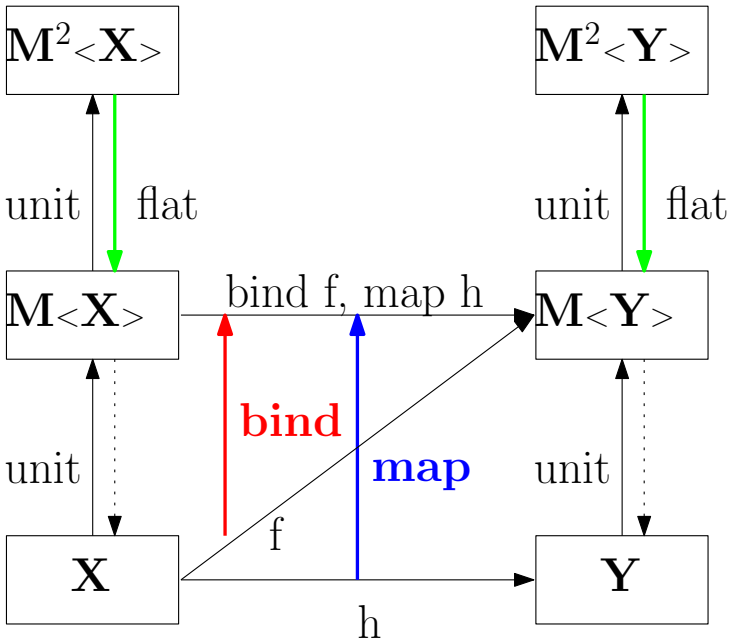
```
1  let add x y = x + y
2  // add : 'T -> 'U -> 'V        // tentative ?
3
4  let a' = add 3 4
5  // add : int -> int -> int    // types are now fixed !
6
7  // let a'' = add 3.0 4.0      // type error !
```

# Statically resolved generic type parameters

- Statically resolved generic type parameter `^T` (caret)

```
1  let inline add x y = x + y
2  // add : ^T -> ^U -> ^V                          // just a template
3  // when (^T ^U): ((+): ^T*^U -> ^V)              // with constraints
4
5  let a' = add 3 4
6  // add : int -> int -> int                       // add for ints
7
8  let a'' = 3.0 + 4.0
9  // add : float -> float -> float                 // add for floats
```

# Monad Zoo – Fundamental Functions



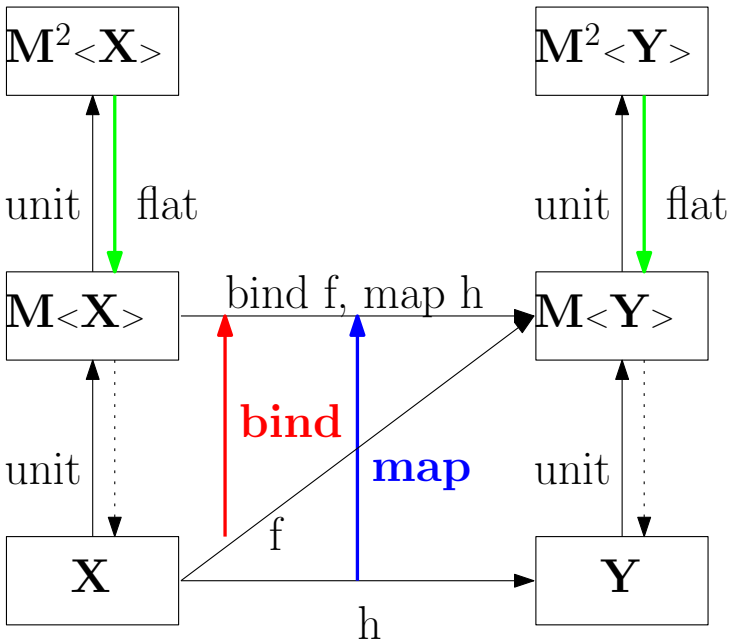

- **unit** : X –> M<X>
- map : (X –> Y) –> (M<X> –> M<Y>)
- bind : (X –> M<Y>) –> (M<X> –> M<Y>)
- flat : M<M<X>> –> M<X>
- Monad laws that make commutative diagrams – later

# Monad Zoo – Fundamental Functions Names

- **unit**, **return**, singleton – cf. related but different **return**!
- map, fmap, Select (LINQ)
- bind, flatmap, collect , SelectMany (LINQ) – cf. **let** !, **do**!
- flat , flatten , concat, **join** (possible ambiguity)
- In category theory, for a monad **M**:
  - map f = **M**f
  - **unit** = $\eta$ (Greek "eta")
  - flat = $\mu$ (Greek "mu")

無

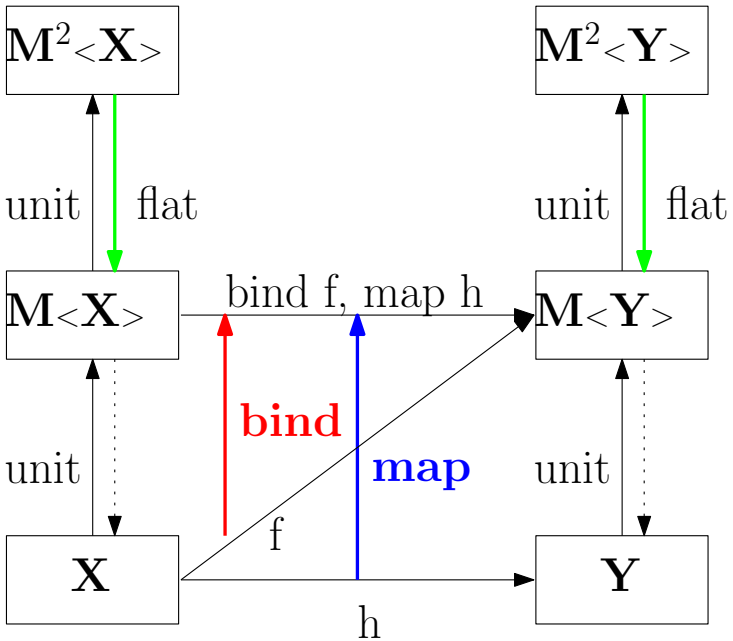
# Monad Zoo – Flat




- flat for lists

```
1 let flat = List.concat
2
3 // [[3;4]; [7;8]] → [3;4;7;8]
```

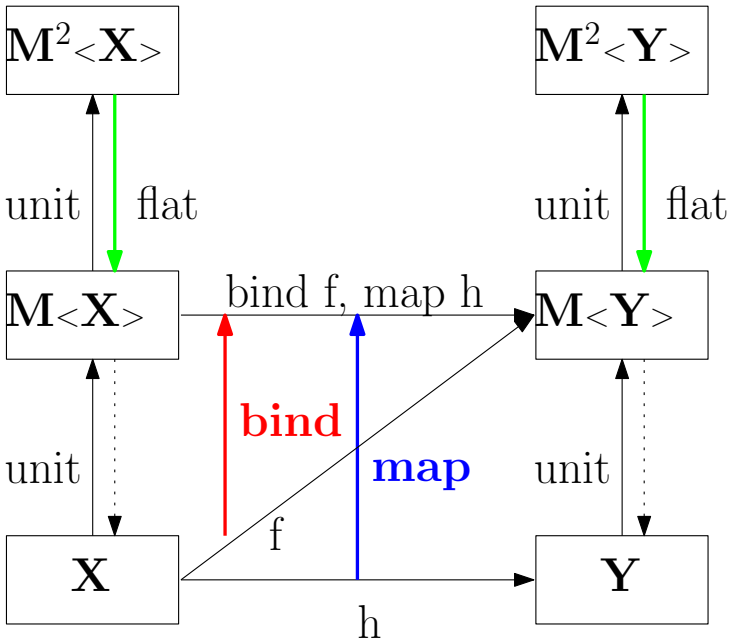
# Monad Zoo – High-Order Bind




- bind for lists aka flatMap

```
1 let flat = List.collect
2
3 let f t = [t; t+1]   // 3 → [3;4]
4 list.collect f       // [3;7] → [3;4;7;8]
5 list.map f           // [3;7] → [[3;4]; [7;8]]
```

# Monad Zoo – Fundamental Functions – "Proofs" later




- FP: unit and bind are fundamental ⇒ map and flat

```
1 let map' h m = bind (h >> unit) m
2 let flat' m2 = bind id m2
```

- Cat: unit, map, and flat are fundamental ⇒ bind

```
1 let bind' f m = map f m |> flat
```

# List Monad Summary – List Module — cf Samples

- Sample test functions

```
1 let h t = t + 1          // h : int -> int
2
3 let f t = [t; t+1]       // f : int -> list<int>
```

- Sample unit (easy from scratch)

```
1 let unit t = [t]
```

# List Monad Summary – List Module — cf Samples

- Definitions using functions from List module

```
1 let map h m = List.map h m
2 map h [3; 7] ⇒ [4; 8]                  // h t = t + 1
3 map f [3; 7] ⇒ [[3;4]; [7;8]]  // f t = [t;t+1]
```

```
1 let bind f m = List.collect f m
2 bind f [3; 7] ⇒ [3; 4; 7; 8]   // f t = [t;t+1]
```

```
1 let flat m2 = List.concat m2
2 flat [[3; 4]; [7; 8]] ⇒ [3; 4; 7; 8]
```

- Alt definitions: (1) from scratch; (2) derived (map ⇐ bind)

# Async Monad – Async Builder — cf Samples

- **async** is a predefined instance of AsyncBuilder

```
1  type AsyncBuilder () =
2
3      member this.Bind (m, f) = ...     // let! do!
4      // m: async<'T>, f: 'T -> Async<'T>
5
6      member this.Return t = ...        // return
7      // t: 'T
8
9      member this.ReturnFrom m = ...    // return!
10     // m: Async<'T>
11
12 let async = AsyncBuilder ()   // new
```

# Async Monad – unit ≡ return, unit1

- **unit** wraps t into an **async**

```
let unit t = async.Return t // X -> Async<X>
let unit t = async {return t}

unit 3 ≡ async {return 3} // 3 -> Async<3>
```

- Sample partial inverse unit1 (to run and get the result)

```
let unit1 =
    Async.RunSynchronously    // Async<X> -> X
```

- Sample expression that runs and gets the result

```
unit1 (unit 3)  // Async<int> -> int
    ≡ unit1 (async {return 3})
    ≡ Async.RunSynchronously (async { return 3 })
    ⇒ 3
```

# Async Monad – id ≡ return!

- Sidebar: the universal id function

```
1  let id x = x  // predefined in the system
2
3  id 3 ⇒ 3
4  id [3;7] ⇒ [3;7]
5  id (async { return t+1 }) ⇒ async { return t+1 }
```

- Interesting case – async call tails : return! ≡ id

```
1  let m = async { return 3 }
2
3  let n = async { return! m } ≡ m ≡ id m
4
5  async.ReturnFrom = id
```

# Async Monad – async.Return, async.ReturnFrom

- Summary so far – poor man's definitions

```
1 asyn.Return t ≡ async { return t }
2
3 async.ReturnFrom m ≡ id m ≡ async { return! t }
```

- Sample functions for our tests

```
1 let h t = t + 1                    // int -> int
2
3 let f t = unit (t+1)               // int -> Async<int>
4 let f t = async { return t+1 }
```

# Async Monad – map, bind ?

- What map should do – simple case

```
1 let h t = t + 1
2
3 async { return 3 } ⟹(map h) async { return 4 }
```

- What if we apply this map ?

```
1 let f t = async { return t+1 }
2
3 async {return 3} ⟹(map f)
4     async {return async {return 4}}
```

- We also get the idea for bind

```
1 async {return 3} ⟹(bind f) async {return 4}
```

# Async Monad – poor man's map

- Poor man's map – logically correct

```
1 let map h m =
2     let t = Async.RunSynchronously m
3     async { return h t }  // async.Return h t
```

- But missing async wait **await** – i.e. **let!**

- Correct but circular definition...

```
1 let map h m = async {
2     let! t = m
3     return h t
4     }
```

# Async Monad – poor man's bind

- Poor man's bind – logically correct

```
1 let bind f m =
2     let t = Async.RunSynchronously m
3     f t
```

- But missing async wait **await** – i.e. **let!**
- Correct but circular definition...

```
1 let bind f m = async {
2     let! t = m
3     return! f t
4     }
```

# Async Monad – Summary

- Fundamental functions

```
1 let unit t = async.Return t =
2     async { return t }
```

```
1 let bind f m = async.Bind (m, f) =
2     async {
3         let! t = m
4         return! f t
5     }
6 // inverted parameter order
```

- We will see actual definitions on our own toy task workflow

# Async Monad – Summary

- Fundamental functions

```
let map h m = bind (h >> unit) m
    async {
        let! t = m
        return h t  // return! async {return h t}
    }
```

```
let flat m2 = bind id m2 =
    async {
        let! t = m2
        return! t  // return! id t
    }
```

- We will see actual definitions on our own toy task workflow

# MyTask – cf Samples

- Goal: An async-like task workflow! A DSL for tasks!

```
1 mytask {
2     let !
3     do!
4     return
5     return!
6 }
```

- For simplicity, with thread blocking waits ☹

# MyTask – Monad Fundamentals

- unit – create a task that will return a given result t

```
1 // unit : 'T -> Task<'T>
2
3 let inline unit t = Task.FromResult t
```

- partial inverse unit1 – wait for m's completion and get result

```
1 // unit1 : Task<'T> -> 'T
2
3 let inline unit1 (m:Task<'T>) = t.Result
4
5 // .Result includes a sync .Wait()
```

# MyTask – Monad Fundamentals

- bind – apply given g to task m

```
1 // bind : ('T -> Task<'U>) -> (Task<'T> -> Task<'U>)
2
3 let inline bind g (m:Task<'T>) =
4     let t = m.Result   // blocking call ☹
5     g t
```

# MyTask – Workflow Builder

- My Workflow Builder Type (Class)

```
1  type MyTaskBuilder () =
2
3    member this.Bind (m, g) =    // let! do!
4      bind g m
5
6    member this.Return t =       // return
7      unit t
8
9    member this.ReturnFrom m =   // return!
10     m
```

- My Workflow Builder Instance

```
1  let mytask = MyTaskBuilder ()
```

# MyTask – unit ≡ **return** ≡ mytask.Return

- Simple Workflow w/ unit ≡ **return** ≡ mytask.Return

```
1 // a : int -> Task<int>
2
3 let a t = mytask {
4     // Thread.Sleep 100
5     return t+1
6     }
7
8 (a 3).Result ⇒ 4
```

```
1 let a t = mytask.Return (t+1)
2
3 let a t = unit (t+1)
```

# MyTask – unit ≡ **return** ≡ mytask.Return

- Simple Workflow w/ unit ≡ **return** ≡ mytask.Return

```
1 // b : int -> Task<int>
2
3 let b t = mytask {
4     // Thread.Sleep 100
5     return t+t
6     }
7 (b 3).Result ⇒ 6
```

```
1 let b t = mytask.Return (t+t)
2
3 let b t = unit (t+t)
```

# MyTask – id ≡ **return!** ≡ mytask.ReturnFrom

- Simple Workflow w/ unit ≡ **return** ≡ mytask.Return

```
1 // a2 : int -> Task<int>
2
3 let a2 t = mytask {
4     // Thread.Sleep 100
5     return! a t
6     }
7 (a2 3).Result ⇒ 4
```

```
1 let a2 t = mytask.ReturnFrom (a t)
2
3 let a2 t = a t = id (a t)
```

# MyTask – bind ≡ **let! do!** ≡ mytask.Bind

- Simple Workflow w/ bind ≡ **let! do!** ≡ mytask.Bind

```
1  // c : int -> Task<int>
2
3  let c t = mytask {   // t = 3
4      let! u = a t     // u = 4
5      return! b u
6      }
7
8  (c 3).Result ⇒ 8
```

```
1  let c t =
2      mytask.Bind (
3          a t,
4          fun u -> mytask.ReturnFrom (b u))
5
6  let c t = bind (fun u -> b u) (a t)
```

# MyTask – Quiz

- Quiz Workflow

```
1 // d : int -> Task<int>
2
3 let d t = mytask {   // t = 3
4     let! u = a t      // u = 4
5     let! v = b u      // v = 8
6     return v
7     }
8
9 (d 3).Result ⇒ 8
```

```
1 let d t = mytask.Bind (...)   //  ???
2
3 let d t = bind ...        // ???
```

# MyTask – Quiz Answers

- Answer #1 – .Bind(m,f) – more useful for workflow semantics

```
1 let d t =
2     mytask.Bind (a t,
3         fun u ->
4             mytask.Bind (b u,
5                 fun v -> mytask.Return v))
```

- Answer #2 – bind f m

```
1 let d t =
2     bind
3         (fun u ->
4             bind (fun v -> unit v) (b u))
5         (a t)
```