

Monads – Part I

Radu Nicolescu
Department of Computer Science
University of Auckland

9 April 2019//10 April 2019

- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions
- ④ Monad Zoo
- ⑤ List Monad
- ⑥ Async Monad
- ⑦ MyTask Monad

Outline

- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions
- ④ Monad Zoo
- ⑤ List Monad
- ⑥ Async Monad
- ⑦ MyTask Monad

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} → async{return 4}`
- Other functions on these types: (flat)
 $[[3;4]; [7;8]] \rightarrow [3;4;7;8]$,
`async{return async{return 3}}` → `async{return 3}`
- Functions that cross boundaries: (unit, unit^{-1})
 $3 \rightarrow [3] \rightarrow 3$, $3 \rightarrow \text{async}\{\text{return } 3\} \rightarrow 3$
- Workflows... `let! do! return return!` ...
- A bit of systematic order? **Monads** come to the rescue!

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!

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})
 $3 \rightarrow [3] \rightarrow 3$, $3 \rightarrow \text{async}\{\text{return } 3\} \rightarrow 3$
- Workflows... let! do! return return! ...
- A bit of systematic order? **Monads** come to the rescue!

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})
 $3 \rightarrow [3] \rightarrow 3$, $3 \rightarrow \text{async{return 3}} \rightarrow 3$
- Workflows... let! do! return return! ...
- A bit of systematic order? **Monads** come to the rescue!

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})
 $3 \rightarrow [3] \rightarrow 3$, $3 \rightarrow \text{async{return 3}} \rightarrow 3$
- Workflows... **let! do! return return!** ...
- A bit of systematic order? **Monads** come to the rescue!

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})
 $3 \rightarrow [3] \rightarrow 3$, $3 \rightarrow \text{async{return 3}} \rightarrow 3$
- 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)
- Specific to **functional programming**
- Formalise the concept of **natural transformation**
- Orthogonal (complementary) to **object-oriented programming**
- We use $M\langle X \rangle$ to represent a generic monad such as `list <int>`, `list <string>`, `async<int>`, `async<string>` ...
- In $F\#$ programming: $X = 'T = ^T = \mathbf{int} = \dots$ as needed

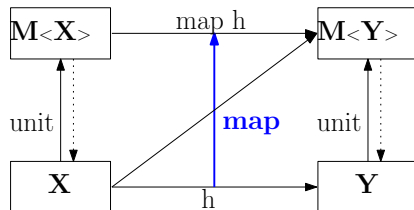
Monads – Informal Introduction

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

Outline

- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions
- ④ Monad Zoo
- ⑤ List Monad
- ⑥ Async Monad
- ⑦ MyTask Monad

Monads – Unit



- Each monad $M\langle X \rangle$ has a “natural” **unit**; e.g., for $M = \text{list}$

```

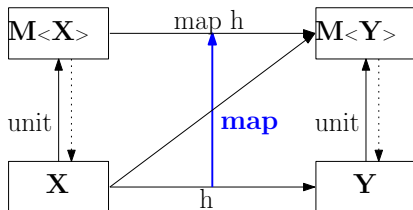
1 unit : 'T -> list <'T> // X -> Y, here X = Y = int
2
3 let inline unit t = [t] // 3 -> [3]
```

- Usually, also a **partial inverse**, unit^{-1} (dotted)

```

1 unit-1 [ t ] = t // [3] -> 3
2
3 unit-1 [], unit-1 [t1, t2] = undefined
```

Monads – Unit



- Each monad $M\langle X \rangle$ has a “natural” **unit**; e.g., for $M = \text{list}$

```

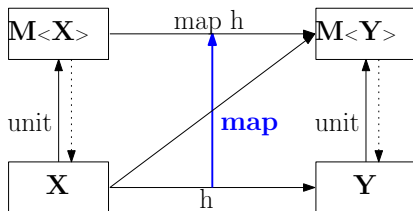
1 unit : 'T -> list <'T> // X -> Y, here X = Y = int
2
3 let inline unit t = [t] // 3 -> [3]
```

- Usually, also a **partial inverse**, unit^{-1} (dotted)

```

1 unit-1 [ t ] = t // [3] -> 3
2
3 unit-1 [], unit-1 [t1, t2] = undefined
```

Monads – High-Order Map



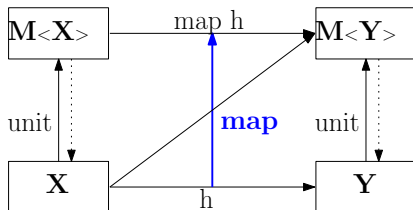
- Each monad $M\langle X \rangle$ has a “natural” high-order **map**

```

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)

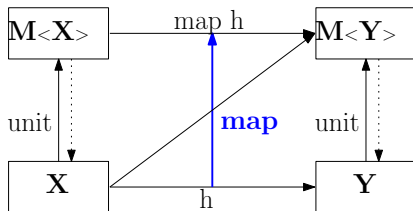
$$1 \quad \mathbf{\text{unit}} \gg \text{map } h \equiv h \gg \mathbf{\text{unit}}$$

- Proofs

$$1 \quad t \xrightarrow{\text{unit}} [t] \xrightarrow{\text{map } h} [t+1]$$

$$1 \quad t \xrightarrow{h} t+1 \xrightarrow{\text{unit}} [t+1]$$

Monads – Diagram Chasing



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

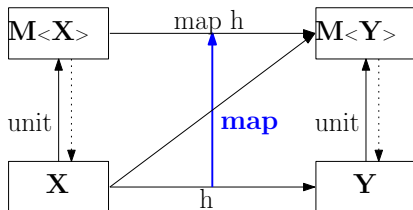
$$1 \quad \mathbf{unit} \gg map\ h \equiv h \gg \mathbf{unit}$$

- Proofs

$$1 \quad t \xrightarrow{unit} [t] \xrightarrow{map\ h} [t+1]$$

$$1 \quad t \xrightarrow{h} t+1 \xrightarrow{unit} [t+1]$$

Monads – Diagram Chasing



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

$$1 \quad \mathbf{unit} \gg \mathbf{map\ h} \equiv \mathbf{h} \gg \mathbf{unit}$$

- Proofs

$$1 \quad t \xrightarrow{\mathbf{unit}} [t] \xrightarrow{\mathbf{map\ h}} [t+1]$$

$$1 \quad t \xrightarrow{\mathbf{h}} t+1 \xrightarrow{\mathbf{unit}} [t+1]$$

Outline

- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions**
- ④ Monad Zoo
- ⑤ List Monad
- ⑥ Async Monad
- ⑦ MyTask Monad

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

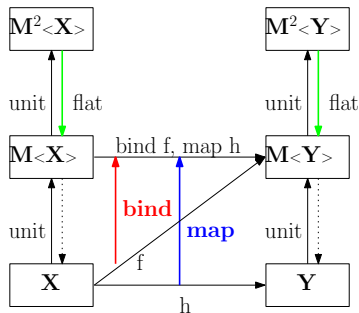
- **Statically resolved** generic type parameter `^T` (caret)

```
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
```

Outline

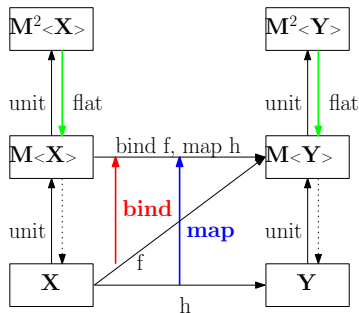
- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions
- ④ Monad Zoo**
- ⑤ List Monad
- ⑥ Async Monad
- ⑦ MyTask Monad

Monad Zoo – Fundamental Functions



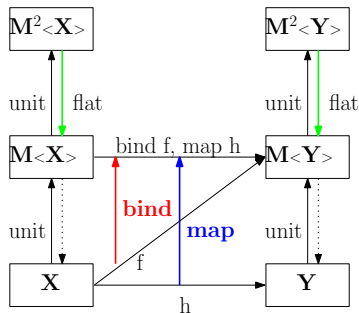
- **unit** : $X \rightarrow M\langle X \rangle$
- map : $(X \rightarrow Y) \rightarrow (M\langle X \rangle \rightarrow M\langle Y \rangle)$
- $bind$: $(X \rightarrow M\langle Y \rangle) \rightarrow (M\langle X \rangle \rightarrow M\langle Y \rangle)$
- $flat$: $M\langle M\langle X \rangle \rangle \rightarrow M\langle X \rangle$
- **Monad laws** that make **commutative diagrams** – later

Monad Zoo – Fundamental Functions



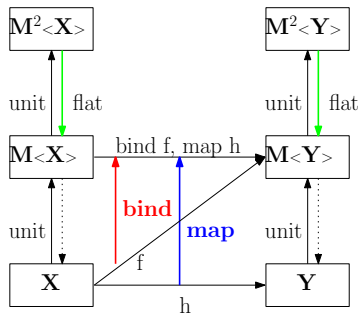
- **unit** : $X \rightarrow M<X>$
- **map** : $(X \rightarrow Y) \rightarrow (M<X> \rightarrow M<Y>)$
- **bind** : $(X \rightarrow M<Y>) \rightarrow (M<X> \rightarrow M<Y>)$
- **flat** : $M<M<X>> \rightarrow M<X>$
- **Monad laws** that make **commutative diagrams** – later

Monad Zoo – Fundamental Functions



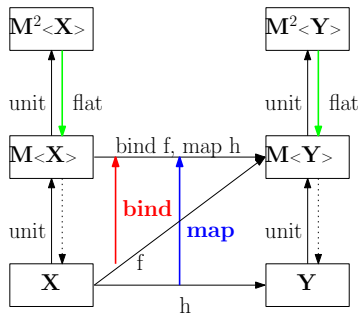
- **unit** : $X \rightarrow M\langle X \rangle$
- **map** : $(X \rightarrow Y) \rightarrow (M\langle X \rangle \rightarrow M\langle Y \rangle)$
- **bind** : $(X \rightarrow M\langle Y \rangle) \rightarrow (M\langle X \rangle \rightarrow M\langle Y \rangle)$
- **flat** : $M\langle M\langle X \rangle \rangle \rightarrow M\langle X \rangle$
- **Monad laws** that make **commutative diagrams** – later

Monad Zoo – Fundamental Functions



- **unit** : $X \rightarrow M\langle X \rangle$
- **map** : $(X \rightarrow Y) \rightarrow (M\langle X \rangle \rightarrow M\langle Y \rangle)$
- **bind** : $(X \rightarrow M\langle Y \rangle) \rightarrow (M\langle X \rangle \rightarrow M\langle Y \rangle)$
- **flat** : $M\langle M\langle X \rangle \rangle \rightarrow M\langle X \rangle$
- **Monad laws** that make **commutative diagrams** – later

Monad Zoo – Fundamental Functions



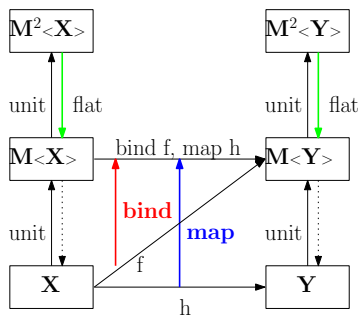
- **unit** : $X \rightarrow M<X>$
- **map** : $(X \rightarrow Y) \rightarrow (M<X> \rightarrow M<Y>)$
- **bind** : $(X \rightarrow M<Y>) \rightarrow (M<X> \rightarrow M<Y>)$
- **flat** : $M<M<X>> \rightarrow M<X>$
- **Monad laws** that make **commutative diagrams** – later

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**:
 - $\text{map } f = \mathbf{M}f$
 - **unit** = η (Greek "eta")
 - flat = μ (Greek "mu")

無

Monad Zoo – Flat

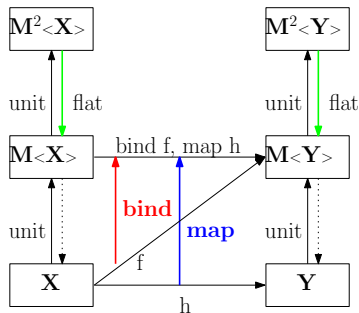


- **flat** for lists

```

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

Monad Zoo – High-Order Bind



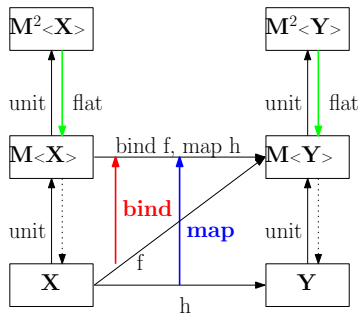
- bind** for lists aka flatMap

```

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 \Rightarrow map and flat

```

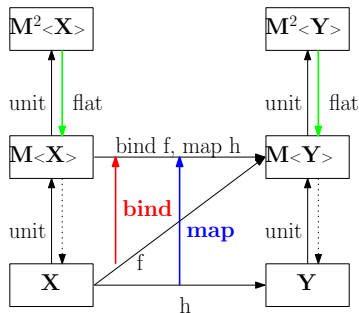
1 let map' h m = bind (h >> unit) m
2 let flat' m2 = bind id m2
  
```

- Cat: unit , map , and flat are fundamental \Rightarrow bind

```

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

Monad Zoo – Fundamental Functions – “Proofs” later



- FP: unit and bind are fundamental \Rightarrow 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 \Rightarrow bind

```

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

Outline

- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions
- ④ Monad Zoo
- ⑤ List Monad**
- ⑥ Async Monad
- ⑦ MyTask Monad

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*

- 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)

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)

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)

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 \Leftarrow bind)

Outline

- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions
- ④ Monad Zoo
- ⑤ List Monad
- ⑥ Async Monad**
- ⑦ MyTask Monad

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` \equiv `return`, `unit1`

- `unit` wraps `t` into an `async`

```
1 let unit t = async.Return t // X -> Async<X>
2 let unit t = async {return t}
3
4 unit 3  $\equiv$  async {return 3} // 3 -> Async<3>
```

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

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

- Sample expression that runs and gets the result

```
1 unit1 (unit 3) // Async<int> -> int
2      $\equiv$  unit1 (async {return 3})
3      $\equiv$  Async.RunSynchronously (async { return 3 })
4      $\Rightarrow$  3
```

Async Monad – `unit` \equiv `return`, `unit1`

- `unit` wraps `t` into an `async`

```
1 let unit t = async.Return t // X -> Async<X>
2 let unit t = async {return t}
3
4 unit 3  $\equiv$  async {return 3} // 3 -> Async<3>
```

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

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

- Sample expression that runs and gets the result

```
1 unit1 (unit 3) // Async<int> -> int
2      $\equiv$  unit1 (async {return 3})
3      $\equiv$  Async.RunSynchronously (async {return 3})
4      $\Rightarrow$  3
```

Async Monad – `unit` \equiv `return`, `unit1`

- `unit` wraps `t` into an `async`

```
1 let unit t = async.Return t // X -> Async<X>
2 let unit t = async {return t}
3
4 unit 3  $\equiv$  async {return 3} // 3 -> Async<3>
```

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

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

- Sample expression that runs and gets the result

```
1 unit1 (unit 3) // Async<int> -> int
2      $\equiv$  unit1 (async {return 3})
3      $\equiv$  Async.RunSynchronously (async { return 3 })
4      $\Rightarrow$  3
```

Async Monad – $\text{id} \equiv \text{return!}$

- Sidebar: the universal **id** function

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

- Interesting case – async call tails : **return!** \equiv **id**

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

Async Monad – `async.Return`, `async.ReturnFrom`

- Summary so far – poor man's definitions

```
1  async.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 – `async.Return`, `async.ReturnFrom`

- Summary so far – poor man's definitions

```
1  async.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 }  $\xRightarrow{\text{map } h}$  async { return 4 }
```

- What if we apply this map ?

```
1 let f t = async { return t+1 }
2
3 async { return 3 }  $\xRightarrow{\text{map } f}$ 
4   async { return async { return 4 } }
```

- We also get the idea for bind

```
1 async { return 3 }  $\xRightarrow{\text{bind } f}$  async { return 4 }
```

Async Monad – map, bind ?

- What map should do – simple case

```
1 let h t = t + 1
2
3 async { return 3 }  $\xRightarrow{\text{map } h}$  async { return 4 }
```

- What if we apply this map ?

```
1 let f t = async { return t+1 }
2
3 async { return 3 }  $\xRightarrow{\text{map } f}$ 
4   async { return async { return 4 } }
```

- We also get the idea for bind

```
1 async { return 3 }  $\xRightarrow{\text{bind } f}$  async { return 4 }
```

Async Monad – map, bind ?

- What map should do – simple case

```
1 let h t = t + 1
2
3 async { return 3 }  $\xRightarrow{\text{map } h}$  async { return 4 }
```

- What if we apply this map ?

```
1 let f t = async { return t+1 }
2
3 async { return 3 }  $\xRightarrow{\text{map } f}$ 
4   async { return async { return 4 } }
```

- We also get the idea for bind

```
1 async { return 3 }  $\xRightarrow{\text{bind } f}$  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

```
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

```
1 let map h m = bind (h >> unit) m
2   async {
3     let! t = m
4     return h t // return! async {return h t}
5   }
```

```
1 let flat m2 = bind id m2 =
2   async {
3     let! t = m2
4     return! t // return! id t
5   }
```

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

Async Monad – Summary

- Fundamental functions

```
1 let map h m = bind (h >> unit) m
2   async {
3     let! t = m
4     return h t // return! async {return h t}
5   }
```

```
1 let flat m2 = bind id m2 =
2   async {
3     let! t = m2
4     return! t // return! id t
5   }
```

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

Outline

- ① Informal Introduction
- ② Unit and High-Order Map
- ③ Inline Functions
- ④ Monad Zoo
- ⑤ List Monad
- ⑥ Async Monad
- ⑦ MyTask Monad

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

- **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 – 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** \equiv **return** \equiv mytask.Return

- Simple Workflow w/ **unit** \equiv **return** \equiv 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** \equiv **return** \equiv mytask.Return

- Simple Workflow w/ **unit** \equiv **return** \equiv 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** \equiv **return** \equiv mytask.Return

- Simple Workflow w/ **unit** \equiv **return** \equiv 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 – **unit** \equiv **return** \equiv mytask.Return

- Simple Workflow w/ **unit** \equiv **return** \equiv 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` \equiv `return!` \equiv `mytask.ReturnFrom`

- Simple Workflow w/ `unit` \equiv `return` \equiv `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 – `id` \equiv `return!` \equiv `mytask.ReturnFrom`

- Simple Workflow w/ `unit` \equiv `return` \equiv `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** \equiv **let!** **do!** \equiv mytask.Bind

- Simple Workflow w/ **bind** \equiv **let!** **do!** \equiv 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 – **bind** \equiv **let!** **do!** \equiv mytask.Bind

- Simple Workflow w/ **bind** \equiv **let!** **do!** \equiv 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

- 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)
```

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)
```