

Monads – Part III

Radu Nicolescu
Department of Computer Science
University of Auckland

4 June 2019

7 June 2019

- ① Revisions
- ② Monad summary
- ③ List Monad
- ④ Option aka Maybe Monad
- ⑤ Homework Suggestions

Outline

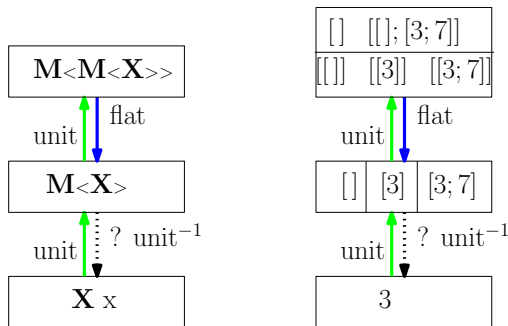
- 1 Revisions
- 2 Monad summary
- 3 List Monad
- 4 Option aka Maybe Monad
- 5 Homework Suggestions

Revision: what is a monad

- In category theory: an endo-functor in the category of types, that formalizes the concept of natural transformations or lifting – NOT in this course
- In practical programming: a systematic uniform way to create and manipulate container-like types
 - Passive containers – just data, no code: lists, arrays, other non-virtual sequences, options (nullables), references, ...
 - Active containers – run code that creates data: virtual sequences, traces, tasks, asyncs, actors, jobs, cloudlets, ...

Revision: unit and flat

- In programming, **unit** creates singleton containers
- unit** cannot create containers that are empty or contain more than one item – thus unit^{-1} is not a function (generally)
- flat** breaks the encapsulation of the double container and removes all inner borders (if any)

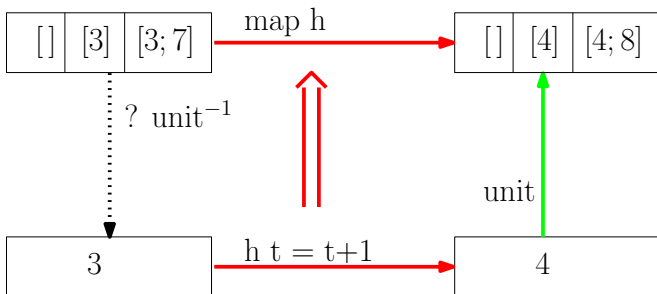


Revision: map horizontal – typical natural lifting

- In programming, **map h** breaks the encapsulation of the container (“unit⁻¹”), applies **h** to each item, and finally recreates a container (unit):

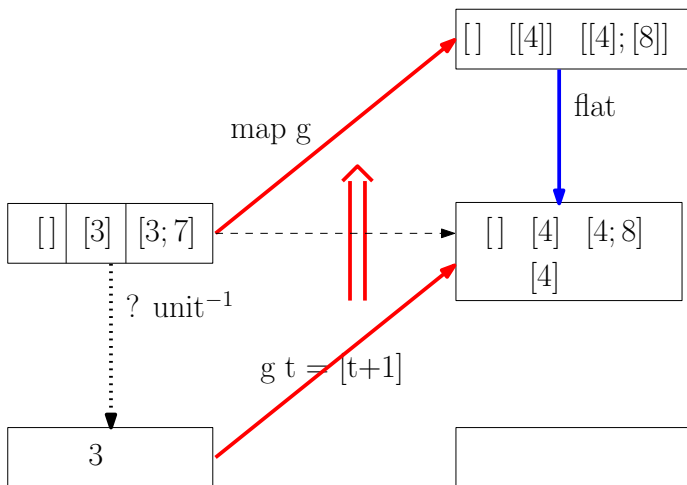
$$1 \quad [3;7] \xrightarrow{\text{map } h} [h \ 3; \ h \ 7]$$

- In the special case when the container is empty, the result is directly empty, bypassing function **h**.



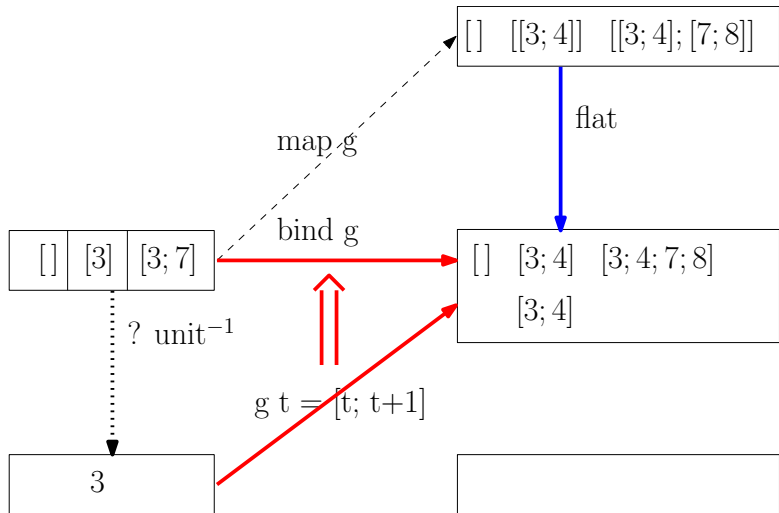
Revision: map diagonal

- **map g**, when **g** is “diagonal”
- **map g >> flat** is **bind**!



Revision: bind (diagonal)

- bind g** is **map g >> flat** ... so it's aka **flatMap**



Revision: homework

- If needed, create such explanatory diagrams for other interesting scenarios, e.g.

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

Outline

- ① Revisions
- ② **Monad summary**
- ③ List Monad
- ④ Option aka Maybe Monad
- ⑤ Homework Suggestions

Monad summary

- Fundamental functions: unit, flat, bind, map
- In programming
 - Base functions: unit, bind
 - Derived functions: flat, map ...
but directly implemented (for efficiency)
 - Three laws for unit and bind
- In category theory
 - Base functions: unit, flat, map
 - Derived function: bind
 - Four laws for unit, flat, map
- In programming we prefer bind, because its relation to
 - sugared monads (`let !`, `do!`)
 - Kleisli operators (`>>=`, `>=>`)

Monad summary

- Fundamental functions: unit, flat, bind, map
- In programming
 - Base functions: unit, bind
 - Derived functions: flat, map ...
but directly implemented (for efficiency)
 - Three laws for unit and bind
- In category theory
 - Base functions: unit, flat, map
 - Derived function: bind
 - Four laws for unit, flat, map
- In programming we prefer bind, because its relation to
 - sugared monads (`let !`, `do!`)
 - Kleisli operators (`>>=`, `>=>`)

Monad summary

- Fundamental functions: unit, flat, bind, map
- In programming
 - Base functions: unit, bind
 - Derived functions: flat, map ...
but directly implemented (for efficiency)
 - Three laws for unit and bind
- In category theory
 - Base functions: unit, flat, map
 - Derived function: bind
 - Four laws for unit, flat, map
- In programming we prefer bind, because its relation to
 - sugared monads (**let!**, **do!**)
 - Kleisli operators ($>>=$, $>=>$)

Outline

- ① Revisions
- ② Monad summary
- ③ List Monad**
- ④ Option aka Maybe Monad
- ⑤ Homework Suggestions

List Monad: unit, and sample functions

- Fundamental functions: unit, flat, bind, map

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

- Sample functions:

```
1 let h t = t+1  
2 let f t = [t]           // f = id >> unit = unit  
3 let f' t = [t+1]       // f = h >> unit  
4 let g t = [t; t+1]    // more complex
```

- Samples:

```
1 h 3 ⇒ 4  
2 f 3 ⇒ [3]  
3 f' 3 ⇒ [4]  
4 g 3 ⇒ [3; 4]
```

List Monad: unit, and sample functions

- Fundamental functions: unit, flat, bind, map

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

- Sample functions:

```
1 let h t = t+1  
2 let f t = [t]           // f = id >> unit = unit  
3 let f' t = [t+1]       // f = h >> unit  
4 let g t = [t; t+1]     // more complex
```

- Samples:

```
1 h 3 ⇒ 4  
2 f 3 ⇒ [3]  
3 f' 3 ⇒ [4]  
4 g 3 ⇒ [3; 4]
```

List Monad: unit, and sample functions

- Fundamental functions: unit, flat, bind, map

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

- Sample functions:

```
1 let h t = t+1  
2 let f t = [t]           // f = id >> unit = unit  
3 let f' t = [t+1]       // f = h >> unit  
4 let g t = [t; t+1]    // more complex
```

- Samples:

```
1 h 3 ⇒ 4  
2 f 3 ⇒ [3]  
3 f' 3 ⇒ [4]  
4 g 3 ⇒ [3; 4]
```

List Monad: flat

- flat (requires at least a double nested list):

```
1  let flat ' m2 = List.concat m2 // F# core lib
2
3  let flat m2 =
4      let mutable r = []
5      for m in m2 do
6          for t in m do
7              r <- r @ [t]
8      r
```

- Samples:

```
1  flat [[3]; []; [5;7]] ⇒ [3; 5; 7]
2  flat [[]] ⇒ []
3  flat [] ⇒ []
```

List Monad: flat

- flat (requires at least a double nested list):

```
1  let flat ' m2 = List.concat m2 // F# core lib
2
3  let flat m2 =
4      let mutable r = []
5      for m in m2 do
6          for t in m do
7              r <- r @ [t]
8      r
```

- Samples:

```
1  flat [[3]; []; [5;7]] ⇒ [3; 5; 7]
2  flat [[]] ⇒ []
3  flat [] ⇒ []
```

List Monad: map

- map (fun h is arbitrary):

```
1 let map' h m = List.map h m // F# core lib
2
3 let map h m =
4     let mutable r = []
5     for t in m do
6         r <- r @ [h t]
7     r
```

- Samples:

```
1 map h [3; 5; 7] ⇒ [4; 6; 8]
2 map h [] ⇒ []
3
4 map g [3; 5; 7] ⇒ [[3; 4]; [5; 6]; [7; 8]]
5 map g [] ⇒ []
```

List Monad: map

- map (fun h is arbitrary):

```
1  let map' h m = List.map h m // F# core lib
2
3  let map h m =
4      let mutable r = []
5      for t in m do
6          r <- r @ [h t]
7      r
```

- Samples:

```
1  map h [3; 5; 7] ⇒ [4; 6; 8]
2  map h [] ⇒ []
3
4  map g [3; 5; 7] ⇒ [[3; 4]; [5; 6]; [7; 8]]
5  map g [] ⇒ []
```

List Monad: bind

- bind (function g returns a list, possibly empty):

```
1  let bind' g m = List.collect g m // F# core lib
2
3  let bind g m =
4      let mutable r = []
5      for t in m do
6          for u in g t do
7              r <- r @ [u]
8      r
```

- Samples:

```
1  bind g [3; 5; 7] ⇒ [3; 4; 5; 6; 7; 8]
2  bind g [] ⇒ []
```

List Monad: bind

- bind (function g returns a list, possibly empty):

```
1  let bind' g m = List.collect g m // F# core lib
2
3  let bind g m =
4      let mutable r = []
5      for t in m do
6          for u in g t do
7              r <- r @ [u]
8      r
```

- Samples:

```
1  bind g [3; 5; 7] ⇒ [3; 4; 5; 6; 7; 8]
2  bind g [] ⇒ []
```

List Monad: Kleisli

- Kleisli operators

```
1 let (>>=) m g = m |> bind g
2
3 let (>=>) f g = f >> bind g
```

- Samples:

```
1 (f >=> g) [3; 5; 7] => [3; 4; 5; 6; 7; 8]
2
3 (f >=> g) [] => []
```

List Monad: Kleisli

- Kleisli operators

```
1 let (>>=) m g = m |> bind g
2
3 let (>=>) f g = f >> bind g
```

- Samples:

```
1 (f >=> g) [3; 5; 7] ⇒ [3; 4; 5; 6; 7; 8]
2
3 (f >=> g) [] ⇒ []
```

List Monad: Builder, Sugared

- List monad builder

```
1 type ListBuilder () =  
2   member this.Bind (m, g) = bind g m // let! do!  
3   member this.ReturnFrom (m) = m      // return!  
4   member this.Return (t) = unit t    // return  
5  
6 let mu = ListBuilder () // new
```

- Sugared

```
1 let x m = mu {  
2   let! t = m  
3   let! u = f t  
4   return! g u  
5 }
```

List Monad: Builder, Sugared

- List monad builder

```
1 type ListBuilder () =  
2   member this.Bind (m, g) = bind g m // let! do!  
3   member this.ReturnFrom (m) = m      // return!  
4   member this.Return (t) = unit t    // return  
5  
6 let mu = ListBuilder () // new
```

- Sugared

```
1 let x m = mu {  
2   let! t = m  
3   let! u = f t  
4   return! g u  
5 }
```

List Monad: Builder, Desugared

- Desugared

```
1 let y m =  
2     mu.Bind (m,  
3         fun t ->  
4             mu.Bind (f t,  
5                 fun u -> mu.ReturnFrom (g u)))
```

- Samples:

```
1 x [3; 5; 7] ≡ y [3; 5; 7] ⇒ [3; 4; 5; 6; 7; 8]  
2  
3 x [] ≡ y [] ⇒ []
```

List Monad: Builder, Desugared

- Desugared

```
1 let y m =  
2     mu.Bind (m,  
3         fun t ->  
4             mu.Bind (f t,  
5                 fun u -> mu.ReturnFrom (g u)))
```

- Samples:

```
1 x [3; 5; 7]  $\equiv$  y [3; 5; 7]  $\Rightarrow$  [3; 4; 5; 6; 7; 8]  
2  
3 x []  $\equiv$  y []  $\Rightarrow$  []
```

Outline

- ① Revisions
- ② Monad summary
- ③ List Monad
- ④ Option aka Maybe Monad
- ⑤ Homework Suggestions

Option Type

- Intuitively, corresponds to a singleton or empty list

```
1 type option <'a> = // 'a option  
2   | Some of 'a // singleton container  
3   | None // empty or null container
```

- It is a tagged container, with zero or at most one item
- Similar to C#.NET **nullable** types, e.g. **int?**
 - Tag **Some** : it contains a value (\approx is NOT “null”)
 - Tag **None** : no value (\approx is “null”)
- Reading: **the billion dollar mistake = null**

Option Type

- Intuitively, corresponds to a singleton or empty list

```
1 type option <'a> = // 'a option
2   | Some of 'a // singleton container
3   | None // empty or null container
```

- It is a tagged container, with zero or at most one item
- Similar to C#.NET **nullable** types, e.g. **int?**
 - Tag **Some** : it contains a value (\approx is NOT “null”)
 - Tag **None** : no value (\approx is “null”)
- Reading: **the billion dollar mistake = null**

Option Type

- Safe model for functions that return null or raise exceptions
- This function seems **int** \rightarrow **int**, but ...

```
1 let divby t = 100 / t
2   divby 5  $\Rightarrow$  20
3   divby 0  $\Rightarrow$  // ? exception
```

- Better function **int** \rightarrow **option**<int>

```
1 let divby t = if t < 0 then Some (100/t) else None
2   divby 5  $\Rightarrow$  Some 20
3   divby 0  $\Rightarrow$  None
```

Option Type

- Safe model for functions that return null or raise exceptions
- This function seems **int** \rightarrow **int**, but ...

```
1 let divby t = 100 / t
2 divby 5  $\Rightarrow$  20
3 divby 0  $\Rightarrow$  // ? exception
```

- Better function **int** \rightarrow **option**<int>

```
1 let divby t = if t < 0 then Some (100/t) else None
2 divby 5  $\Rightarrow$  Some 20
3 divby 0  $\Rightarrow$  None
```

Option Monad: unit, and sample functions

- Fundamental functions: unit, flat, bind, map

```
1 let unit t = Some t
```

- Sample functions:

```
1 let h t = t+1  
2 let f t = Some t // f = id >> unit = unit  
3 let f' t = Some (t+1) // f' = h >> unit  
4 let g t = Some (t+1) // g = h >> unit
```

- Samples:

```
1 h 3 ⇒ 4  
2 f 3 ⇒ Some 3  
3 f' 3 ⇒ Some 4  
4 g 3 ⇒ Some 4
```

Option Monad: unit, and sample functions

- Fundamental functions: unit, flat, bind, map

```
1 let unit t = Some t
```

- Sample functions:

```
1 let h t = t+1  
2 let f t = Some t // f = id >> unit = unit  
3 let f' t = Some (t+1) // f' = h >> unit  
4 let g t = Some (t+1) // g = h >> unit
```

- Samples:

```
1 h 3 ⇒ 4  
2 f 3 ⇒ Some 3  
3 f' 3 ⇒ Some 4  
4 g 3 ⇒ Some 4
```

Option Monad: unit, and sample functions

- Fundamental functions: unit, flat, bind, map

```
1 let unit t = Some t
```

- Sample functions:

```
1 let h t = t+1  
2 let f t = Some t // f = id >> unit = unit  
3 let f' t = Some (t+1) // f' = h >> unit  
4 let g t = Some (t+1) // g = h >> unit
```

- Samples:

```
1 h 3 ⇒ 4  
2 f 3 ⇒ Some 3  
3 f' 3 ⇒ Some 4  
4 g 3 ⇒ Some 4
```

Option Monad: flat

- flat (requires at least a double nested option):

```
1  let flat ' m2 =  
2      if Option.isSome m2 then Option.get m2  
3      else None  
4  
5  let flat m2 =  
6      match m2 with  
7      | Some m -> m  
8      | None -> None
```

- Samples:

```
1  flat (Some (Some 3)) ⇒ Some 3  
2  flat (Some None) ⇒ None  
3  flat None ⇒ None
```

Option Monad: flat

- flat (requires at least a double nested option):

```
1  let flat ' m2 =
2      if Option.isSome m2 then Option.get m2
3      else None
4
5  let flat m2 =
6      match m2 with
7      | Some m -> m
8      | None -> None
```

- Samples:

```
1  flat (Some (Some 3)) ⇒ Some 3
2  flat (Some None) ⇒ None
3  flat None ⇒ None
```

Option Monad: map

- map (fun h is arbitrary):

```
1 let map' h m = Option.map h m // F# core lib
2
3 let map h m =
4     match m with
5     | Some t -> h t |> unit
6     | None -> None
```

- Samples:

```
1 map h (Some 3) => Some 4
2 map h None => None
3
4 map g (Some 3) => Some (Some 4)
5 map g None => None
```

Option Monad: map

- map (fun h is arbitrary):

```
1 let map' h m = Option.map h m // F# core lib
2
3 let map h m =
4     match m with
5     | Some t -> h t |> unit
6     | None -> None
```

- Samples:

```
1 map h (Some 3) ⇒ Some 4
2 map h None ⇒ None
3
4 map g (Some 3) ⇒ Some (Some 4)
5 map g None ⇒ None
```

Option Monad: bind

- bind (function g returns an option, possibly None):

```
1  let bind' g m = Option.bind g m
2
3  let bind g m =
4      match m with
5      | Some t -> g t
6      | None -> None
```

- Samples:

```
1  bind g (Some 3) ⇒ Some 4
2  bind g None ⇒ None
```

Option Monad: bind

- bind (function g returns an option, possibly None):

```
1  let bind' g m = Option.bind g m
2
3  let bind g m =
4      match m with
5      | Some t -> g t
6      | None -> None
```

- Samples:

```
1  bind g (Some 3) ⇒ Some 4
2  bind g None ⇒ None
```

Option Monad: Kleisli

- Kleisli operators

```
1 let (>>=) m g = m |> bind g
2
3 let (>=>) f g = f >> bind g
```

- Samples:

```
1 (f >=> g) (Some 3) => Some 4
2
3 (f >=> g) None => None
```

Option Monad: Kleisli

- Kleisli operators

```
1 let (>>=) m g = m |> bind g
2
3 let (>=>) f g = f >> bind g
```

- Samples:

```
1 (f >=> g) (Some 3) => Some 4
2
3 (f >=> g) None => None
```

Option Monad: Builder, Sugared

- Option monad builder

```
1 type OptionBuilder () =  
2   member this.Bind (m, g) = bind g m // let! do!  
3   member this.ReturnFrom (m) = m      // return!  
4   member this.Return (t) = unit t    // return  
5  
6 let mu = OptionBuilder () // new
```

- Sugared

```
1 let x m = mu {  
2   let! t = m  
3   let! u = f t  
4   return! g u  
5 }
```

Option Monad: Builder, Sugared

- Option monad builder

```
1 type OptionBuilder () =  
2   member this.Bind (m, g) = bind g m // let! do!  
3   member this.ReturnFrom (m) = m      // return!  
4   member this.Return (t) = unit t    // return  
5  
6 let mu = OptionBuilder () // new
```

- Sugared

```
1 let x m = mu {  
2   let! t = m  
3   let! u = f t  
4   return! g u  
5 }
```

Option Monad: Builder, Desugared

- Desugared

```
1 let y m =  
2     mu.Bind (m,  
3         fun t ->  
4             mu.Bind (f t,  
5                 fun u -> mu.ReturnFrom (g u)))
```

- Samples:

```
1 x (Some 3) ≡ y (Some 3) ⇒ Some 4  
2  
3 x None ≡ y None ⇒ None
```

Option Monad: Builder, Desugared

- Desugared

```
1 let y m =  
2     mu.Bind (m,  
3         fun t ->  
4             mu.Bind (f t,  
5                 fun u -> mu.ReturnFrom (g u)))
```

- Samples:

```
1 x (Some 3) ≡ y (Some 3) ⇒ Some 4  
2  
3 x None ≡ y None ⇒ None
```

Outline

- ① Revisions
- ② Monad summary
- ③ List Monad
- ④ Option aka Maybe Monad
- ⑤ Homework Suggestions

Homework Suggestions

- Consolidate your understanding by repeating similar exercises with other monads
- Arrays
- Sequences
- Task – cf. MyTask
- Async – hard!
- Trace
- References