

## Monads – Part II

Radu Nicolescu  
Department of Computer Science  
University of Auckland

9 April 2019//10 April 2019//4 June 2019

① LINQ Workflows vs. F# Workflows

② Kleisli Operators

③ Monad Laws

# LINQ defines Problem Specific Workflows

- C# Query Comprehensions
  - on sequences or sequence-like queryable collections

```
1 var x = new[] { 10, 20, 30, };
2
3 var y = from t in x
4         where t%4 != 0 // → .Where (...)
5         select t+1;    // → .Select (...)
```

- LINQ: Query Comprehensions → **Enumerable** Methods

```
1 var y = x.Where(t => t%4 != 0).Select (t => t+1);
```

```
1 var y = Enumerable.Select (
2     Enumerable.Where(x, t => t%4 != 0),
3     t => t+1);
```

# F# Workflows

- F# pipeline – **List** module functions

```
1 let x = [ 10; 20; 30; ]
2
3 let y = x |> List.filter (fun t -> t%4 <> 0)
4           |> List.map (fun t -> t+1)
```

- F# Query Comprehensions → **QueryBuilder** Methods

```
1 let y = query {
2     for t in x do           // → .For (...)
3     where (t%4 <> 0)       // → .Where (...)
4     select (t+1)           // → .Select (...)
5 }
```

https:

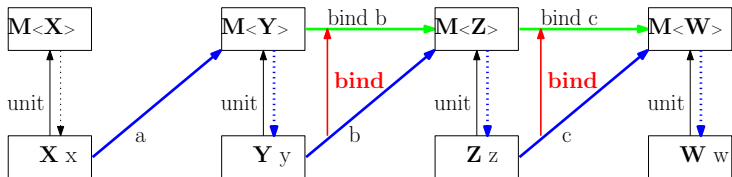
[//msdn.microsoft.com/en-us/visualfsharpdocs/  
conceptual/linq.querybuilder-class-%5Bfsharp%5D](https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/linq.querybuilder-class-%5Bfsharp%5D)

# F# Workflows

- F# are highly **customizable**
- F# are more **powerful** and cover a wide variety of **domains**:  
seq, query, async, actor, job, cloud, ...  
<http://mbrace.io/programming-model.html>

```
1  let first = cloud { return 15 }
2  let second = cloud { return 27 }
3
4  cloud {
5      let! x = first
6      let! y = second
7      return x + y
8  }
```

# Composing Diagonal Arrows

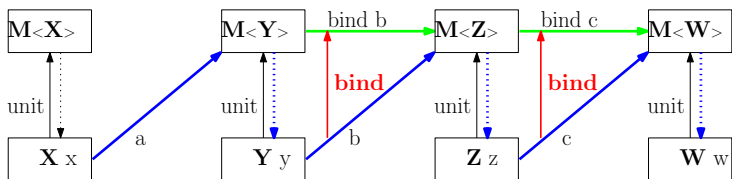


- How to combine several diagonal arrows – cf. the blue line

```
1  let a x = mytask { return x+1 }  
2  let b y = mytask { return y+y }  
3  let c z = mytask { return z*z }
```

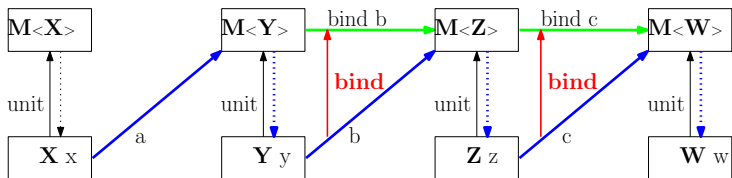
```
1  x = 3 ⇒ y = 4 ⇒ z = 8 ⇒ w = 64
```

## Solution: Workflow



```
1 let d x = mytask { // x = 3
2   let! y = a x // mytask.Bind (a x, ...) // y = 4
3   let! z = b y // mytasks.Bind (b y, ...) // z = 8
4   return! c z // mytask.ReturnFrom (c z) // unit 64
5 }
6 let w = unit1 (d 3)
```

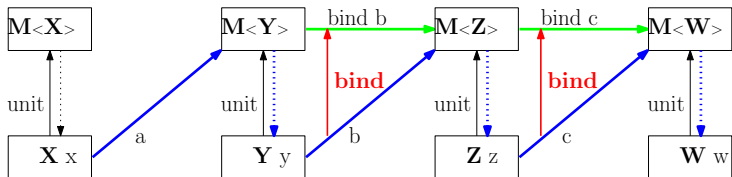
# Solution: Builder Methods



```
1 let d x =  
2   mytask.Bind (a x,  
3     fun y ->  
4       mytask.Bind (b y,  
5         fun z -> mytask.ReturnFrom c z))
```

```
1     fun z -> c z))  
2   c))
```

# Solution: Monad Functions



```
1 let d x = bind (fun y -> bind c (b y)) (a x)
2
3 let d x = a x |> bind (fun y -> (bind c << b) y)
4 let d x = a x |> bind (bind c << b)
5 let d x = a x |> bind (b >> bind c)
6
7 let d x = x |> a >> bind (b >> bind c)
```

Note the patterns: `m |> bind f` and `f >> bind g`

# An Even “Better” Solution ? Kleisli Operators!

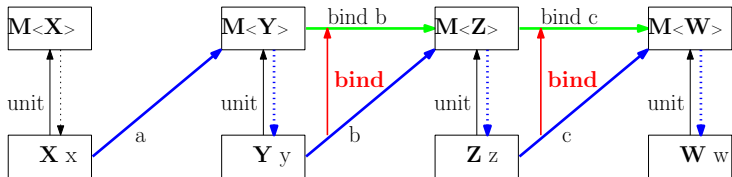
- Bind operator

```
1 // (>>=) : Task<'T> -> ('T->Task<'U>)  
2 //           -> Task<'U>  
3  
4 let inline (>>=) m f = bind f m // m |> (bind f)
```

- Kleisli composition – aka Kleisli “fish” or “komposition”

```
1 // (>=>) : ('T->Task<'U>) -> ('U->Task<'V>)  
2 //           -> ('T->Task<'V>)  
3  
4 let inline (>=>) f g = f >> (bind g)
```

# Solution: Kleisli Composition!



```
1 let k x = x |> (a >=> (b >=> c))
```

As we will see, ( $\text{>=>}$ ) is associative, so

```
1 let k x = x |> ((a >=> b) >=> c)
```

```
2 let k x = x |> (a >=> b >=> c)
```

Hopac “komposition”:

<https://hopac.github.io/Hopac/Hopac.html>

```
1 (>=>): ('x->#Job<'y>) -> ('y->#Job<'z>) -> 'x->Job<'z>
```

# Three Laws for Classical Function Composition

- **Left identity** – id

$$1 \quad \text{id} \gg g \equiv g \quad // \quad 0 + g = g, \quad 1 * g = g$$

- **Right identity** – id

$$1 \quad f \gg \text{id} \equiv f \quad // \quad g + 0 = g, \quad g * 1 = g$$

- **Associativity**

$$1 \quad (f \gg g) \gg h \equiv f \gg (g \gg h) \quad // \quad (f+g)+h = f+(g+h)$$

- **Note: NOT Commutative** – in general

$$1 \quad f \gg g \not\equiv g \gg f$$

# Three Monad Laws – w/ w/- Kleisli

- **Left identity** – unit

```
1 unit >> bind g ≡ g
2
3 unit >=> g ≡ g
```

- **Right identity** – unit

```
1 bind unit ≡ id
2
3 f >=> unit ≡ f
```

- **Associativity**

```
1 bind g >> bind h ≡ bind (g >> bind h)
2
3 (f >=> g) >=> h ≡ f >=> (g >=> h)
```

## Three Monad Laws – Why ?

- Diagrams that are **naturally** expected to be **commutative**
- **Anecdotal evidence** for simple monads, such as **List**; recall:

```
1 bind g [u;v] ≡ flat (map g [u;v])
2                ≡ flat ([g u; g v])
```

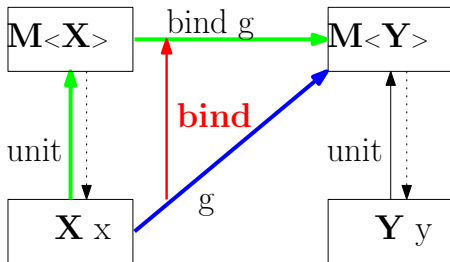
```
1 let g t = [t+1]
2 map g [3; 7] ≡ [g 3; g 7] ≡ [[4]; [8]]
3 flat (map g [3; 7]) ≡ flat [[4]; [8]] ≡ [4; 8]
4 bind g [3; 7] ≡ [4; 8]
```

- **Actual proof** in the special case when **unit is invertible** (later)

```
1 (unit >> unit-1) t = t
2 (unit-1 >> unit) m = m
```

- **Naturally** expected **workflow transformations** (later)

Left identity :  $\text{unit} \gg \text{bind } g \equiv g$



# Left identity : $\text{unit} \gg \text{bind } g \equiv g$

- Consider for example:

```
1 let g t = [t; t+1]
```

- Evaluate left-side on a given t:

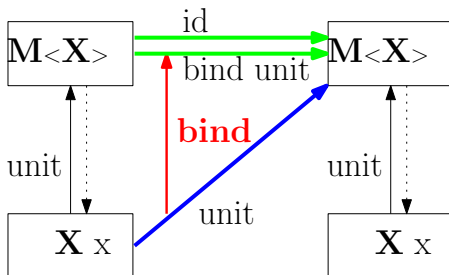
```
1 (unit >> bind g) t  
2   ≡ bind g (unit t)  
3   ≡ bind g [t] ≡ flat (map g [t])  
4   ≡ flat [g t] ≡ flat [[t; t+1]]  
5   ≡ [t; t+1]  
6   ≡ g t
```

- Thus:

```
1 unit >> bind g ≡ g
```

- Cf. definition of ( $\gg$ ):

```
1 unit >=> g ≡ g
```

Right identity :  $\text{bind unit} \equiv \text{id}$ 

## Right identity : `bind unit ≡ id`

- Evaluate left-side on a given list, e.g. `m = [3; 7]`:

```
1 (bind unit) m
2   ≡ bind unit [3; 7] ≡ flat (map unit [3; 7])
3   ≡ flat [unit 3; unit 7] ≡ flat [[3]; [7]]
4   ≡ [3; 7]
5   ≡ id m
```

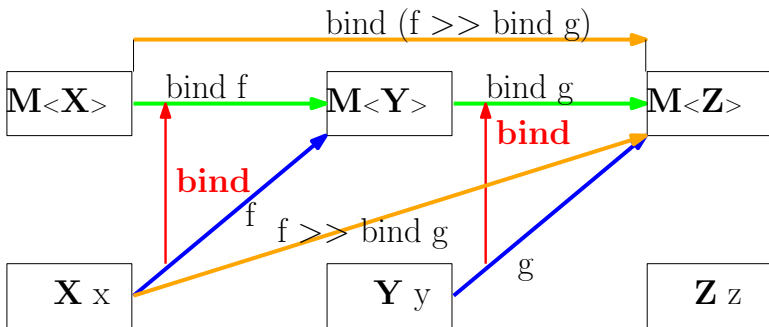
- Thus:

```
1 bind unit ≡ id
```

- Combine with an arbitrary `f` on the left:

```
1 f >> (bind unit) ≡ f >> id
2 f >=> unit ≡ f
```

Associativity :  $\text{bind } g \gg \text{bind } h \equiv \text{bind } (g \gg \text{bind } h)$



# Associativity : $\text{bind } g \gg \text{bind } h \equiv \text{bind } (g \gg \text{bind } h)$

- Consider for example:

```
1 let g t = [t; t+1]
2 let h t = [t+2]
```

- Evaluate left-side on a given list, e.g.  $m = [3;7]$ :

```
1 (bind g >> bind h) [3;7]
2   ≡ bind h (bind g [3;7])
3   ≡ bind h (flat [g 3; g 7])
4   ≡ bind h (flat [[3;4]; [7;8]])
5   ≡ bind h [3; 4; 7; 8]
6   ≡ flat [h 3; h 4; h 7; h 8]
7   ≡ flat [[5]; [6]; [9]; [10]]
8   ≡ [5; 6; 9; 10]
```

# Associativity : $\text{bind } g \gg \text{bind } h \equiv \text{bind } (g \gg \text{bind } h)$

- Recall:

```
1 let g t = [t; t+1]
2 let h t = [t+2]
```

- Evaluate right-side on a given list, e.g.  $m = [3;7]$ :

```
1 bind (g >> bind h) [3;7]
2   ≡ flat [(g >> bind h) 3; (g >> bind h) 7]
3   ≡ flat [bind h (g 3); bind h (g 7)]
4   ≡ flat [bind h [3;4]; bind h [7;8]]
5   ≡ flat [flat [h 3; h 4]; flat [h 7; h 8]]
6   ≡ flat [flat [[5]; [6]]; flat [[9]; [10]]]
7   ≡ flat [[5;6]; [9;10]]
8   ≡ [5; 6; 9; 10]
```

# Associativity : $\text{bind } g \gg \text{bind } h \equiv \text{bind } (g \gg \text{bind } h)$

- Thus:

```
1  bind g >> bind h ≡ bind (g >> bind h)
```

- Combine with an arbitrary f on the left:

```
1  f >> (bind g >> bind h) ≡ f >> bind (g >> bind h)
2
3  (f >> bind g) >> bind h ≡ f >> bind (g >> bind h)
4
5  (f >=> g) >=> h ≡ f >=> (g >=> h)
```

# What we get for map?

- Recall:

```
1 let map h = bind (h >> unit)
```

- Left identity:

```
1 unit >> bind g ≡ g  
2 unit >> bind (h >> unit) ≡ h >> unit  
3 unit >> map h ≡ h >> unit
```

- Right identity:

```
1 bind unit ≡ id  
2 bind (id >> unit) ≡ id  
3 map id ≡ id
```