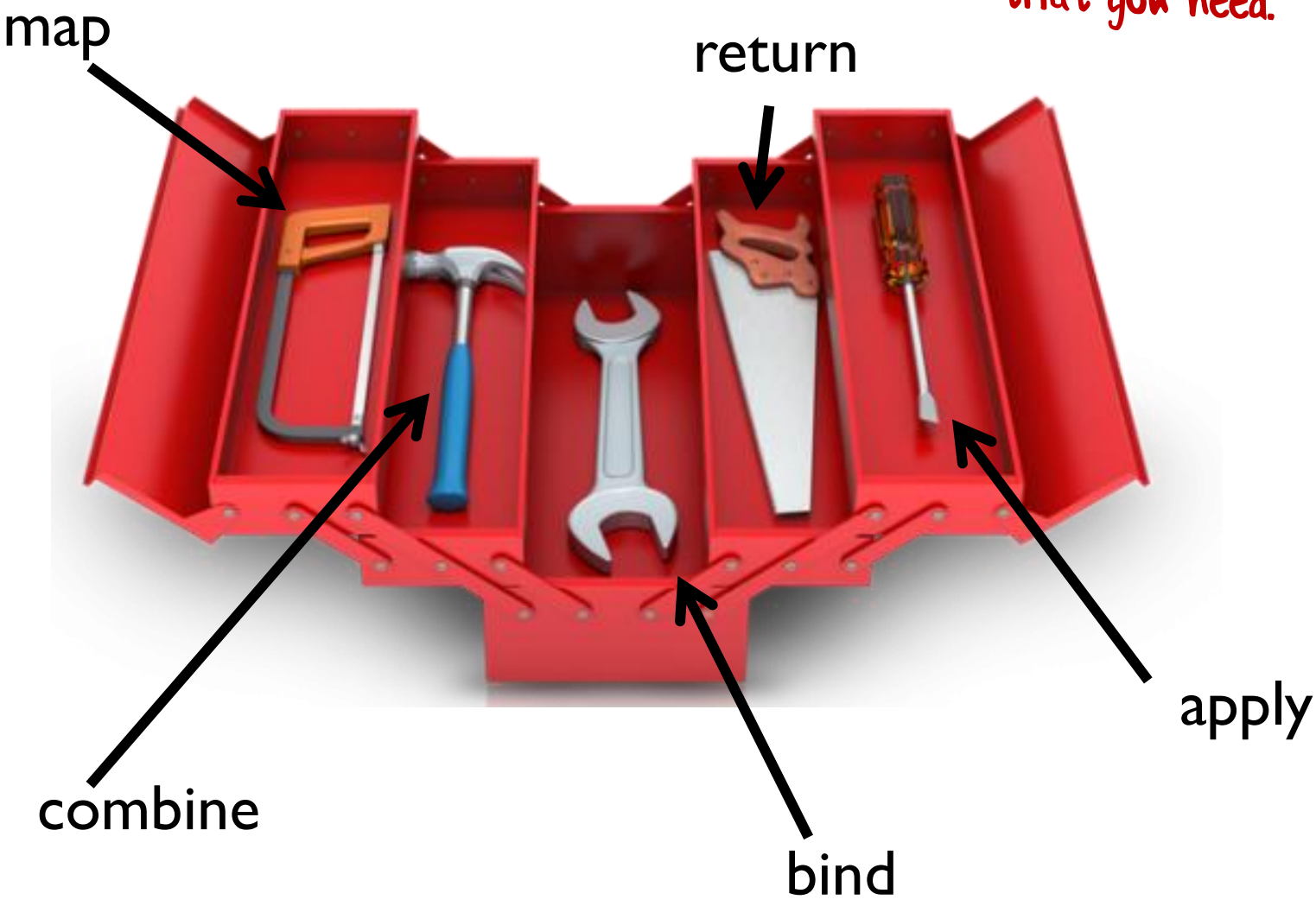
The Functional Programming Toolbox

A.k.a. Introducing Monads

The Functional Toolbox

About 11 important tools that you need.



The Functional Toolbox

is for problem-solving!

- Composition
- Combination/Aggregation
- Iteration
- Working with effects
  - Mixing effects and non-effects
  - Chaining effects in series
  - Working with effects in parallel
  - Pulling effects out of a list

The Functional Toolbox

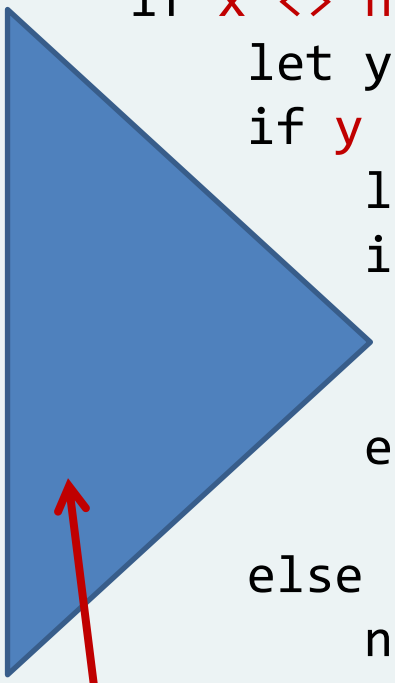
- Composition: `compose`
- Iteration: `fold`
- Combination/Aggregation: `combine` & `reduce`
- Working with effects
  - Mixing effects and non-effects: `map` & `return`
  - Chaining effects in series: `bind/flatMap`
  - Working with effects in parallel: `apply` & `zip`
  - Pulling effects out of a list: `sequence` & `traverse`

Functional Toolbox (FP jargon version)

- Combination/Aggregation: `Monoid`
- Working with effects
  - Mixing effects and non-effects: `Functor`
  - Chaining effects in series: `Monad`
  - Working with effects in parallel: `Applicative`

Motivation:
Taming the "pyramid of doom"

```
let example input =
    let x = doSomething input
    if x <> null then
        let y = doSomethingElse x
        if y <> null then
            let z = doAThirdThing y
            if z <> null then
                let result = z
                result
            else
                null
        else
            null
    else
        null
```

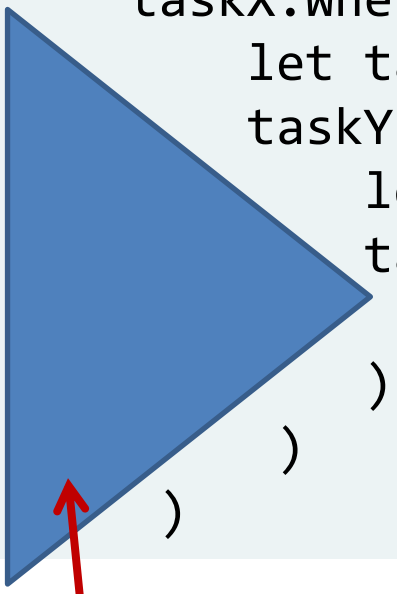


Nested null checks

The "pyramid of doom"

I know you could do early returns, but bear with me...

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                z  // final result
            )
        )
    )
```



Nested callbacks

Another "pyramid of doom"

Let's fix this!

```
let example input =
    let x = doSomething input
    if x <> null then
        let y = doSomethingElse x
        if y <> null then
            let z = doAThirdThing y
            if z <> null then
                let result = z
                result
            else
                null
        else
            null
    else
        null
```

Nulls are a code smell:
replace with Option!

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                let result = z.Value
                Some result
            else
                None
        else
            None
    else
        None
```

Much more elegant, yes?

No! This is fugly!

But there is a pattern we can exploit...

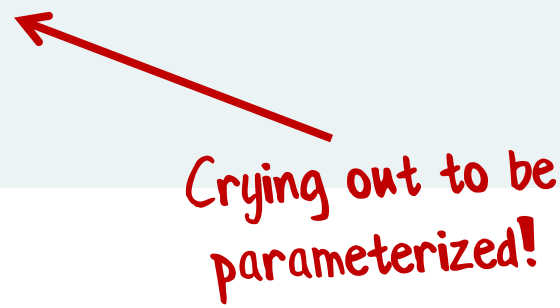
```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                // do something with z.Value
                // in this block
            else
                None
        else
            None
    else
        None
```

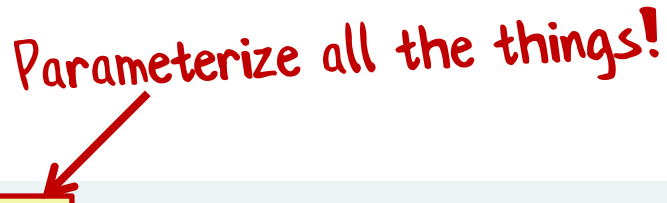
```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            // do something with y.Value
            // in this block



        else
            None
    else
        None
```

```
let example input =
    let x = doSomething input
    if x.IsSome then
        // do something with x.Value
        // in this block




    else
        None
```

Can you see the pattern?

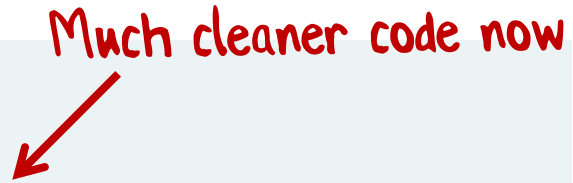
```
if opt.IsSome then
    //do something with opt.Value
else
    None
```





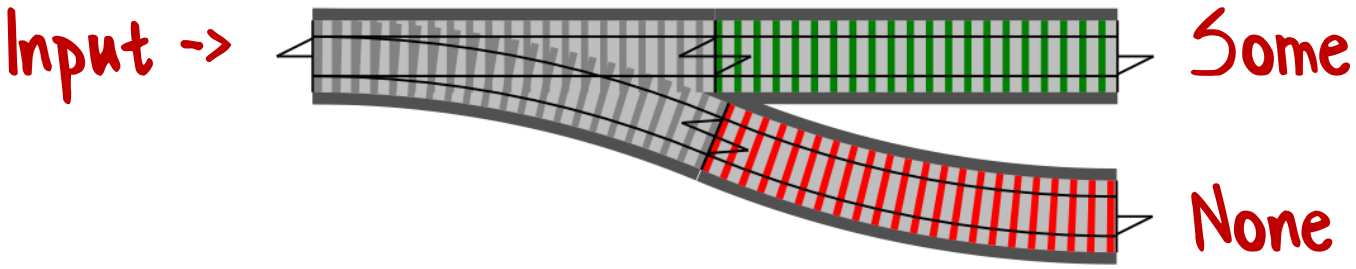

```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```

```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```




```
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo ...
```

This is an example of a more general problem



Pattern:
Use bind to chain options

Before

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                let result = z.Value
                Some result
            else
                None
        else
            None
    else
        None
```

After

```
let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

Same as "ifSomeDo"

After

```
let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

```
let example input =
    doSomething input
    |> bind doSomethingElse
    |> bind doAThirdThing
    |> bind ...
```

No pyramids!

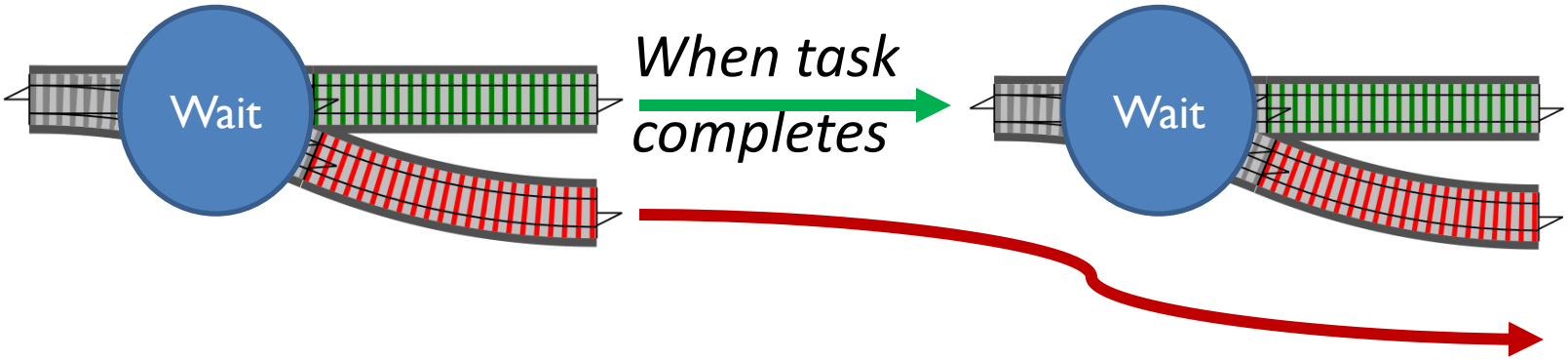
Code is linear and clear.

This pattern is called "monadic bind"

Pattern:

Use bind to chain tasks

a.k.a "promise" "future"


Wait
When task
completes
Wait

Before

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                z  // final result
            )
        )
    )
```

After

```
let taskBind f task =
    task.WhenFinished (fun taskResult ->
        f taskResult)

let taskExample input =
    startTask input
    |> taskBind startAnotherTask
    |> taskBind startThirdTask
    |> taskBind ...
```

This pattern is also a "monadic bind"

The F# async type

Async in F# is a *type*

- `"fetchUrl"`
  - `string -> Async<string>`
- `"loadCustomer"`
  - `customerId -> Async<Customer>`

Async in F# is a *type*

- Combine them using bind
- Or async computation expression

Understanding "effects"

What is an effect?

Could be anything really. It's vague!

- A generic type

  `List<_>`

- A type enhanced with extra data

  `Option<_>, Result<_>`

- A type that can change the outside world

  `Async<_>, Task<_>, Random<_>`

- A type that carries state

  `State<_>, Parser<_>`

What is an effect?

We'll focus on three for this talk

- A generic type

  `List<_>`
- A type enhanced with extra data

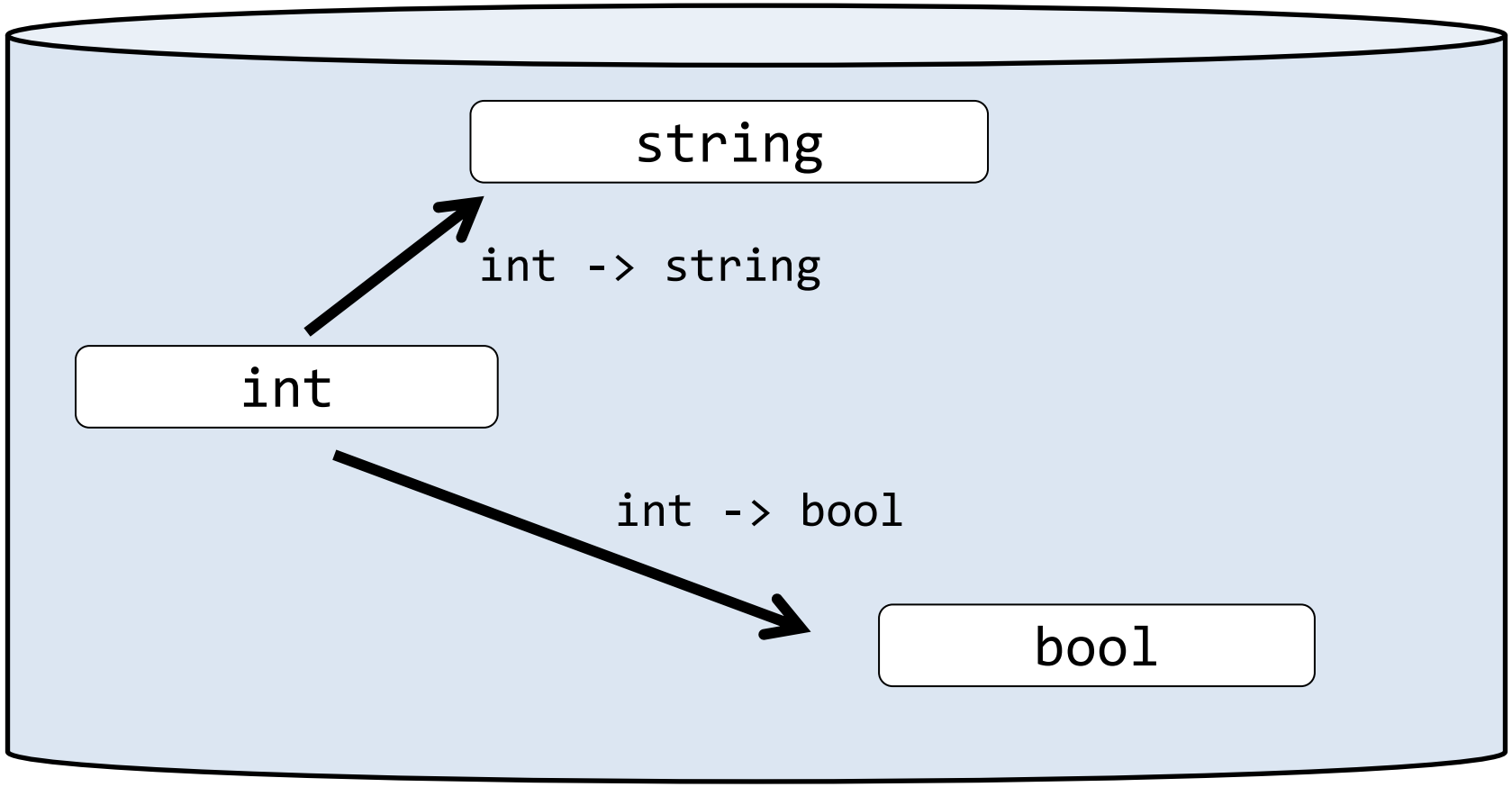
  `Option<_>`, `Result<_>`
- A type that can change the outside world

  `Async<_>`, `Task<_>`, `Random<_>`
- A type that carries state

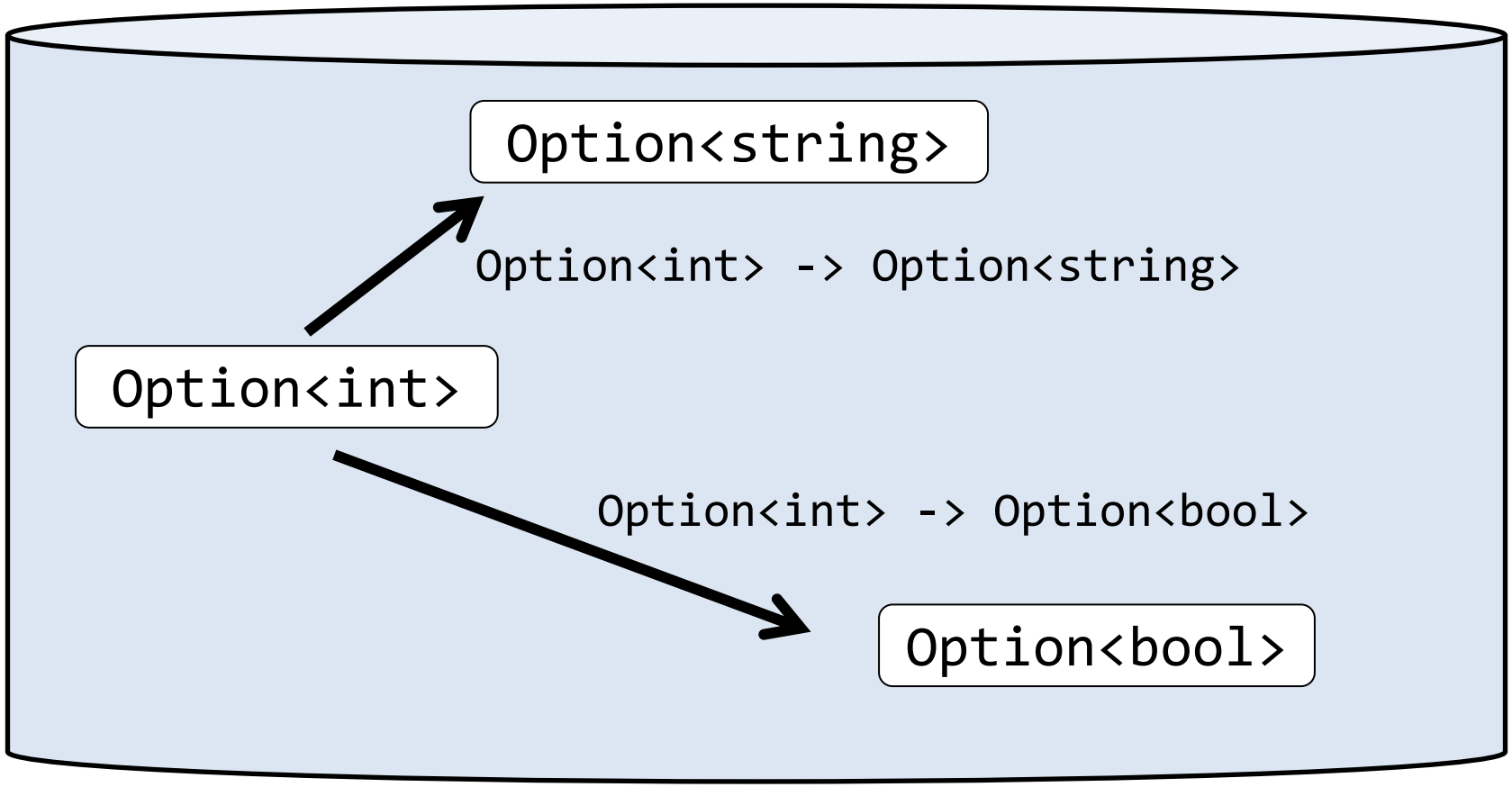
  `State<_>`, `Parser<_>`

"Normal" world vs. "Effects" world

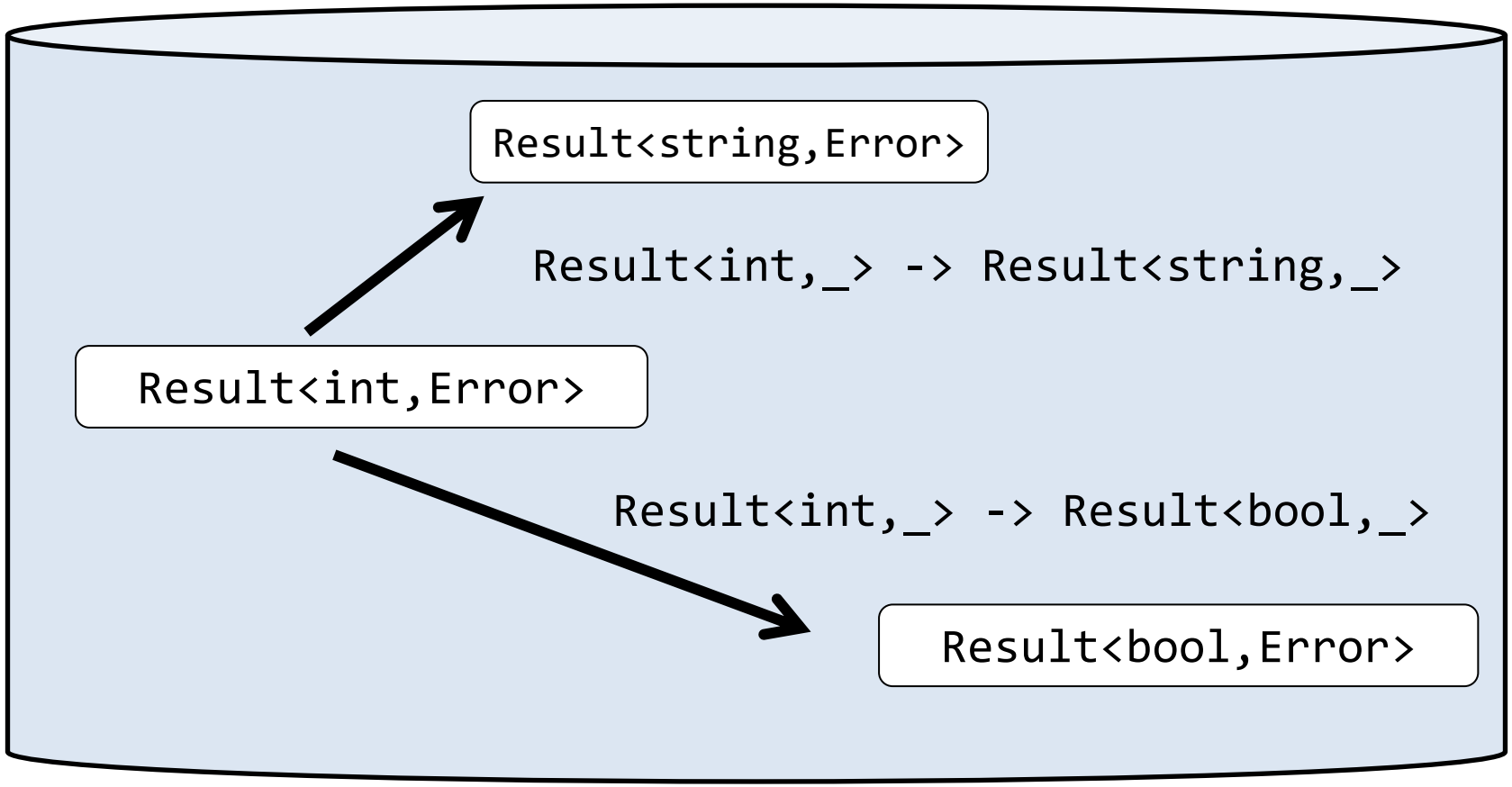
"Normal" world



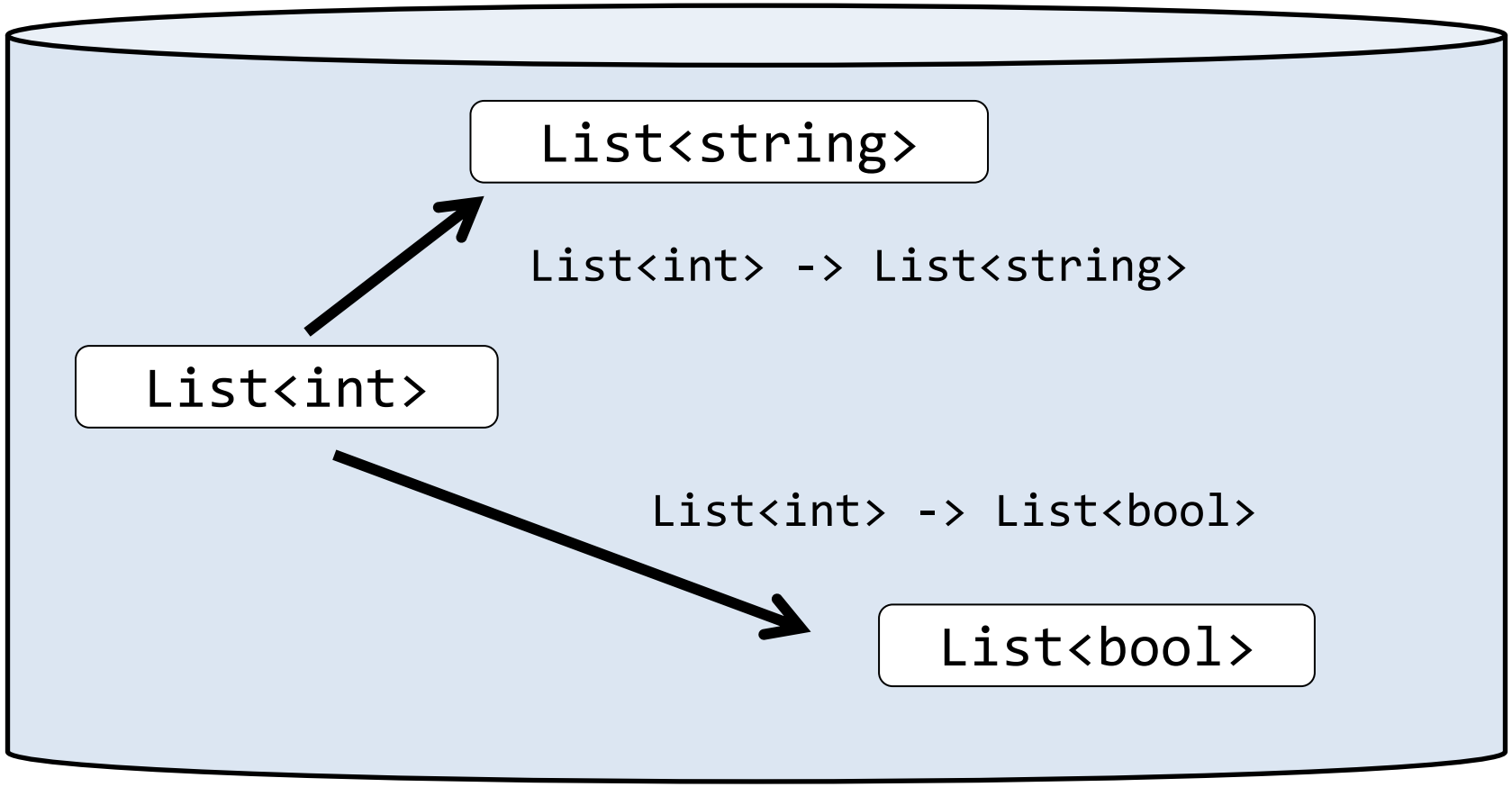
"Option" world



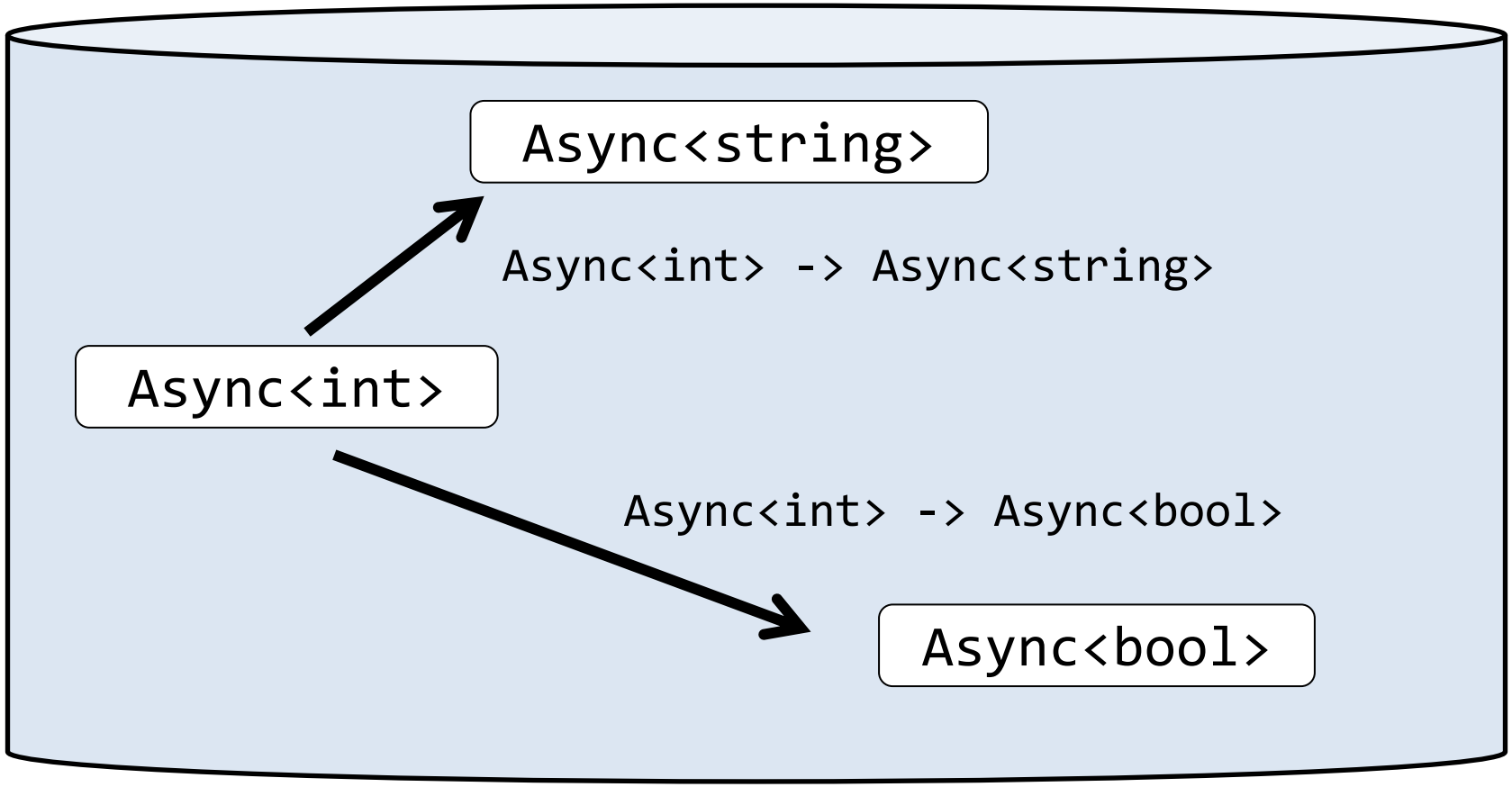
"Result" world



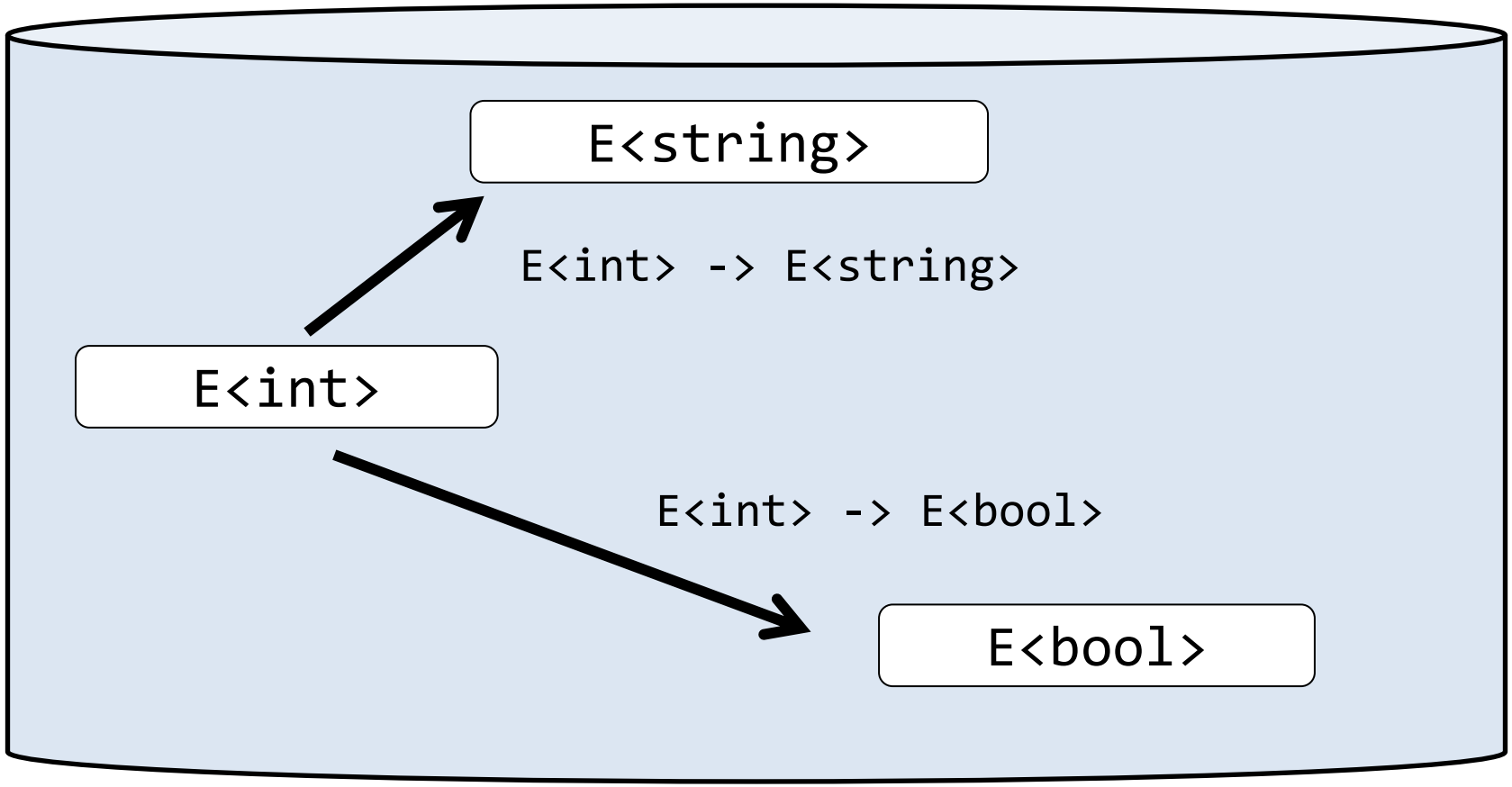
"List" world


List<string>
List<int> -> List<string>
List<int>
List<int> -> List<bool>
List<bool>

"Async" world


Async<string>
Async<int> -> Async<string>
Async<int>
Async<int> -> Async<bool>
Async<bool>

"Effects" world



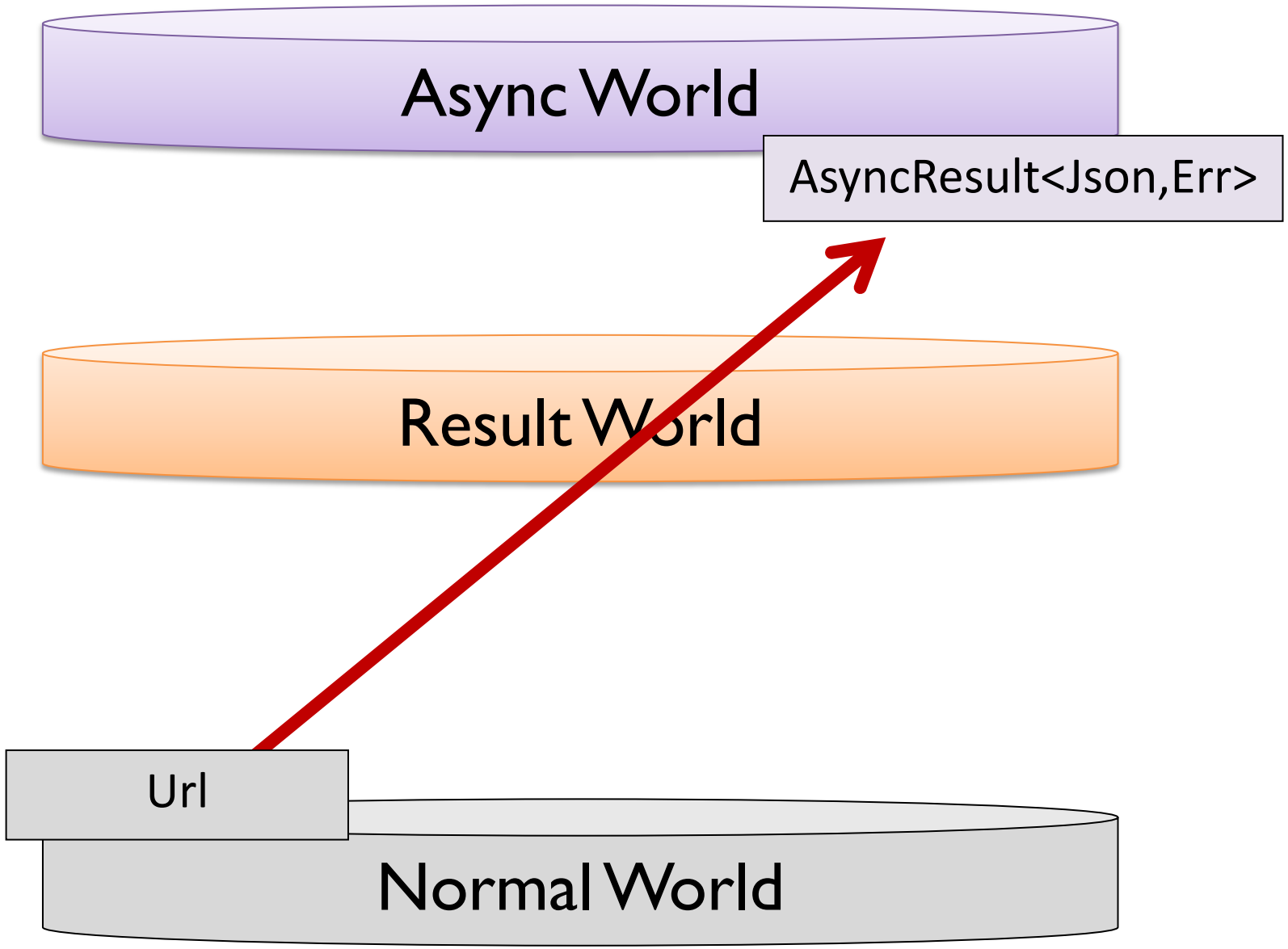
Problem:

How to do stuff in an effects world?

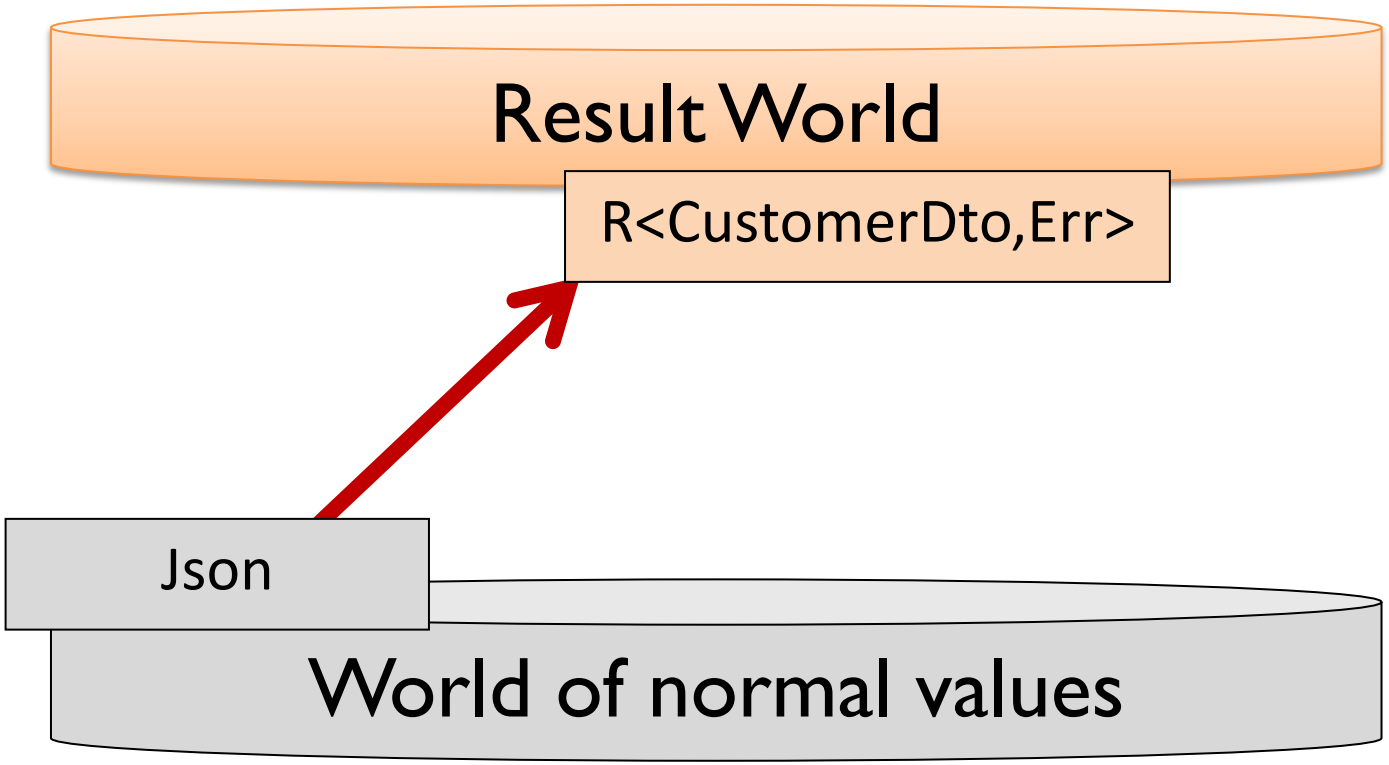
Example scenario

- Download a URL into a JSON file
- Decode the JSON into a Customer DTO
- Convert the DTO into a valid Customer
- Store the Customer in a database

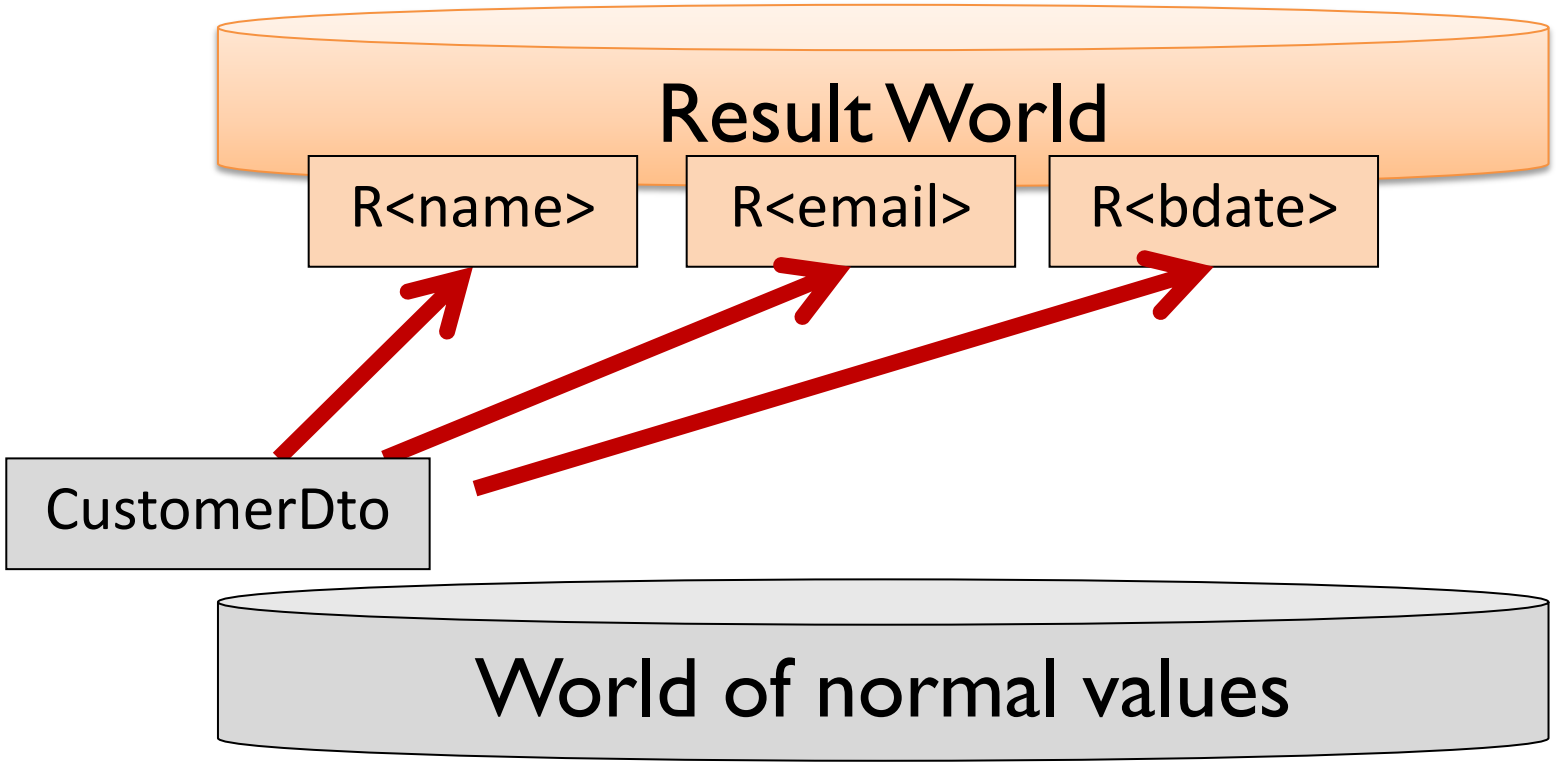
Download the json file



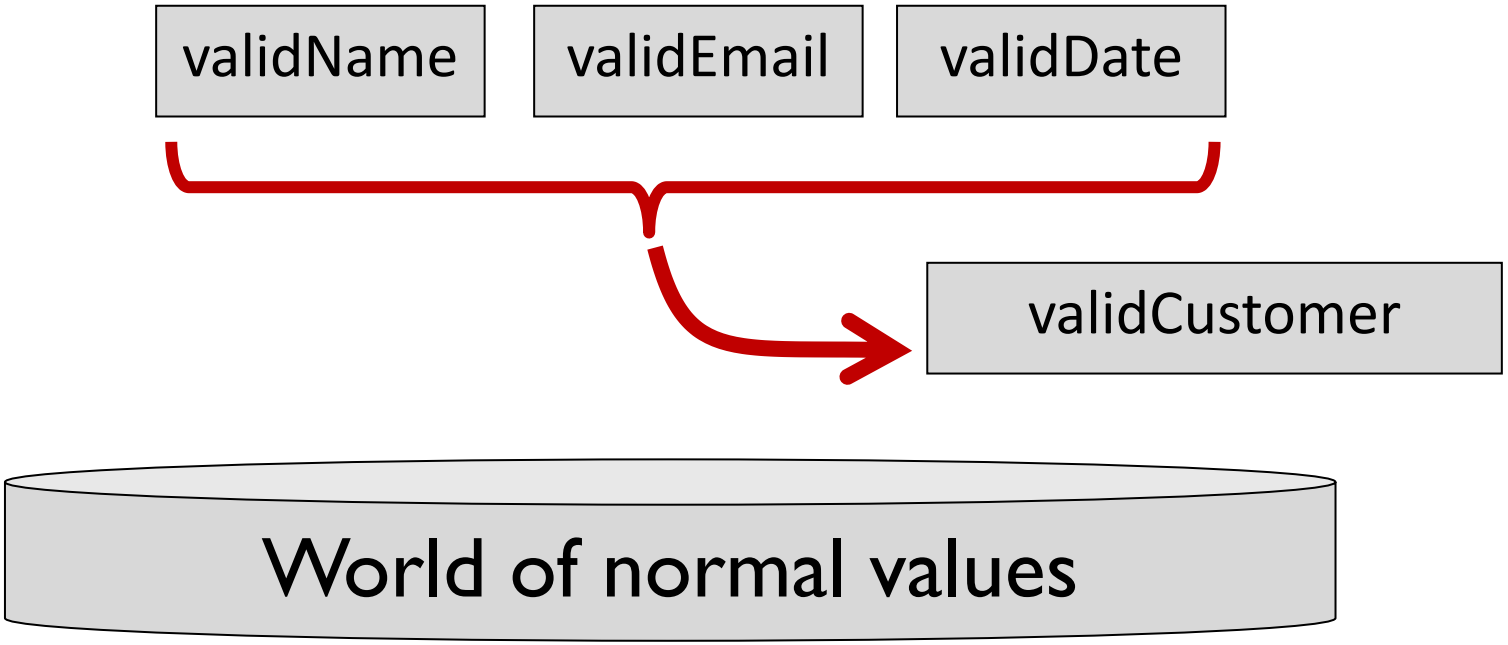
Decode the json



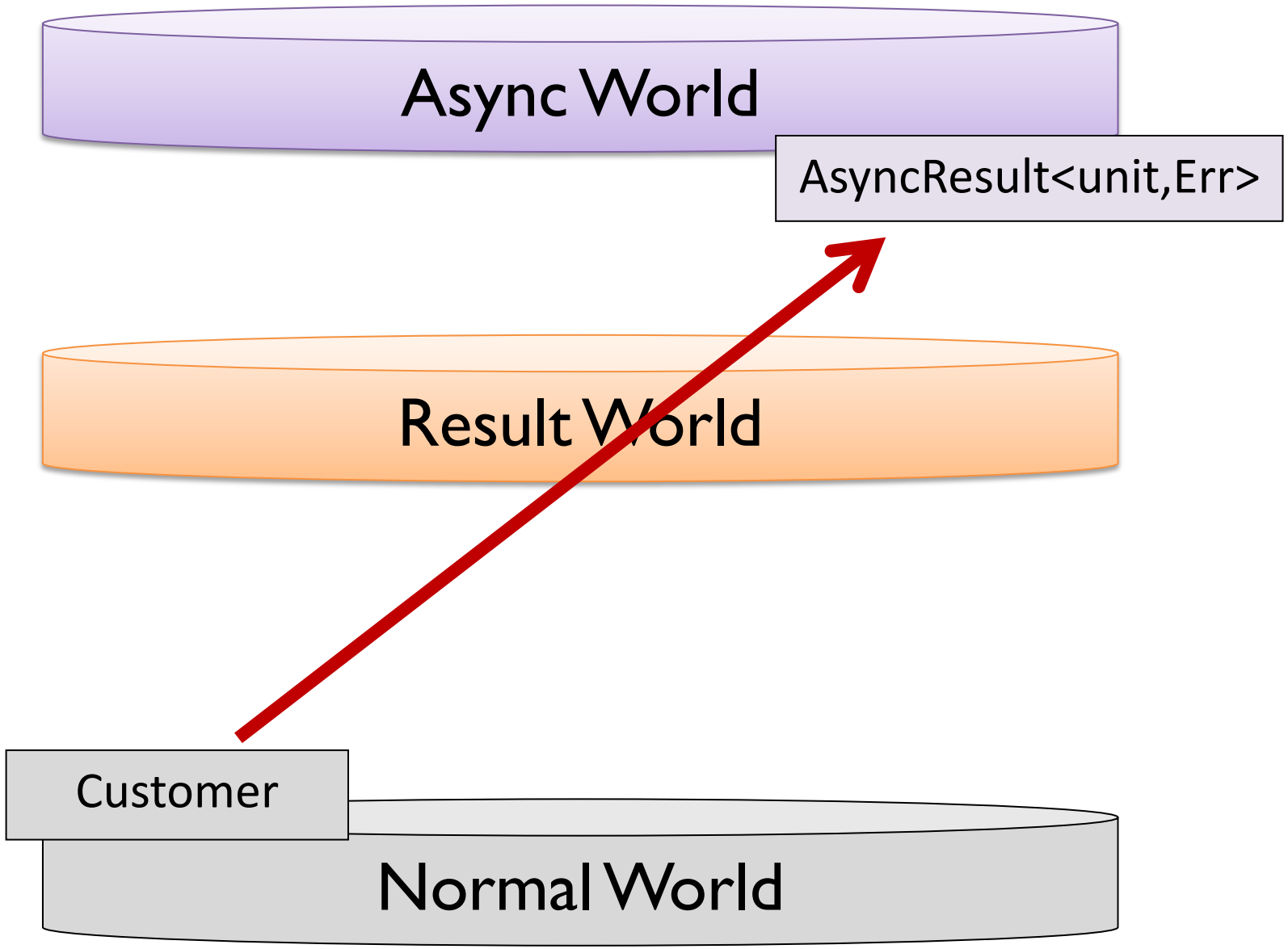
Validate fields



Construct the customer



Store the customer

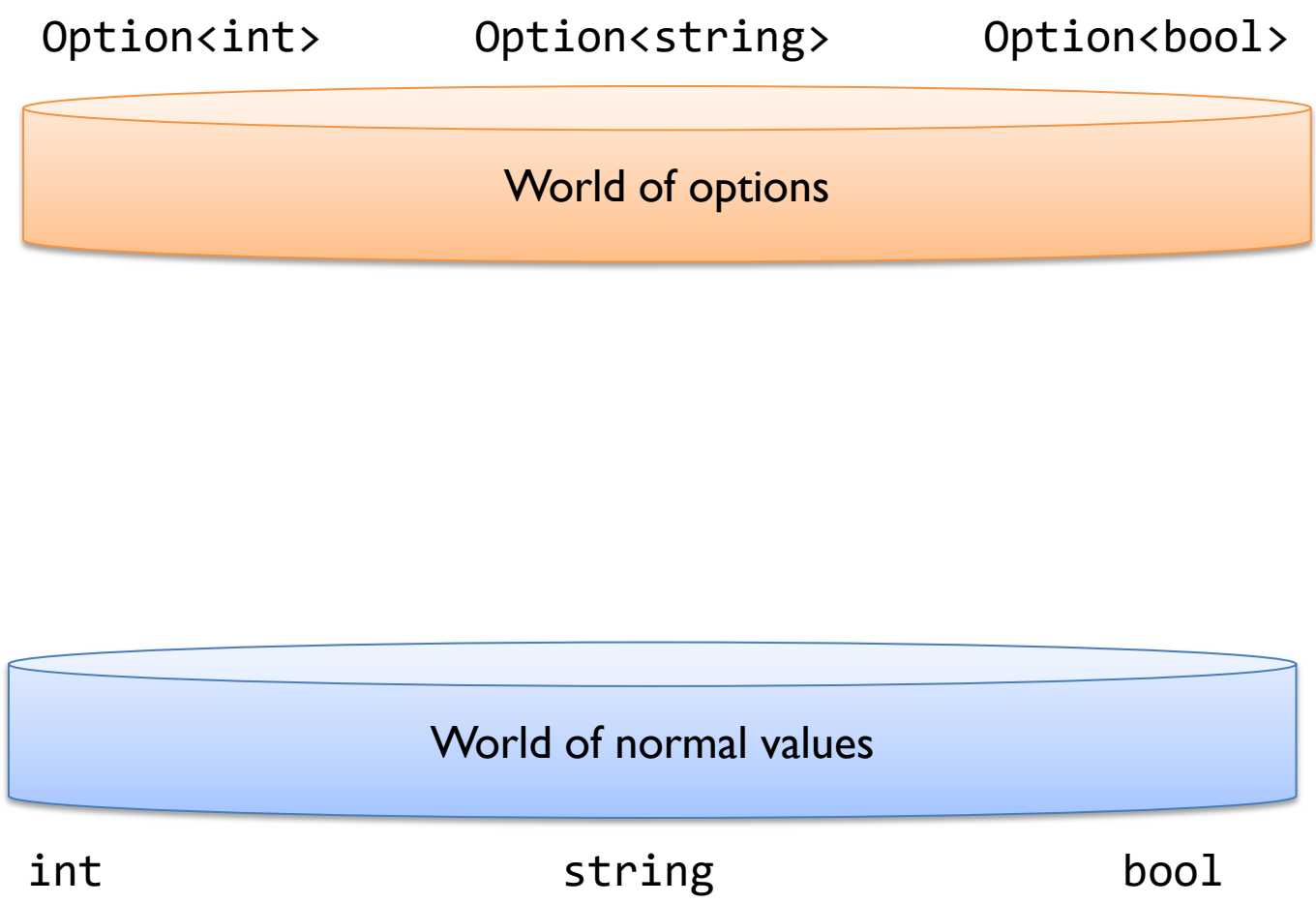


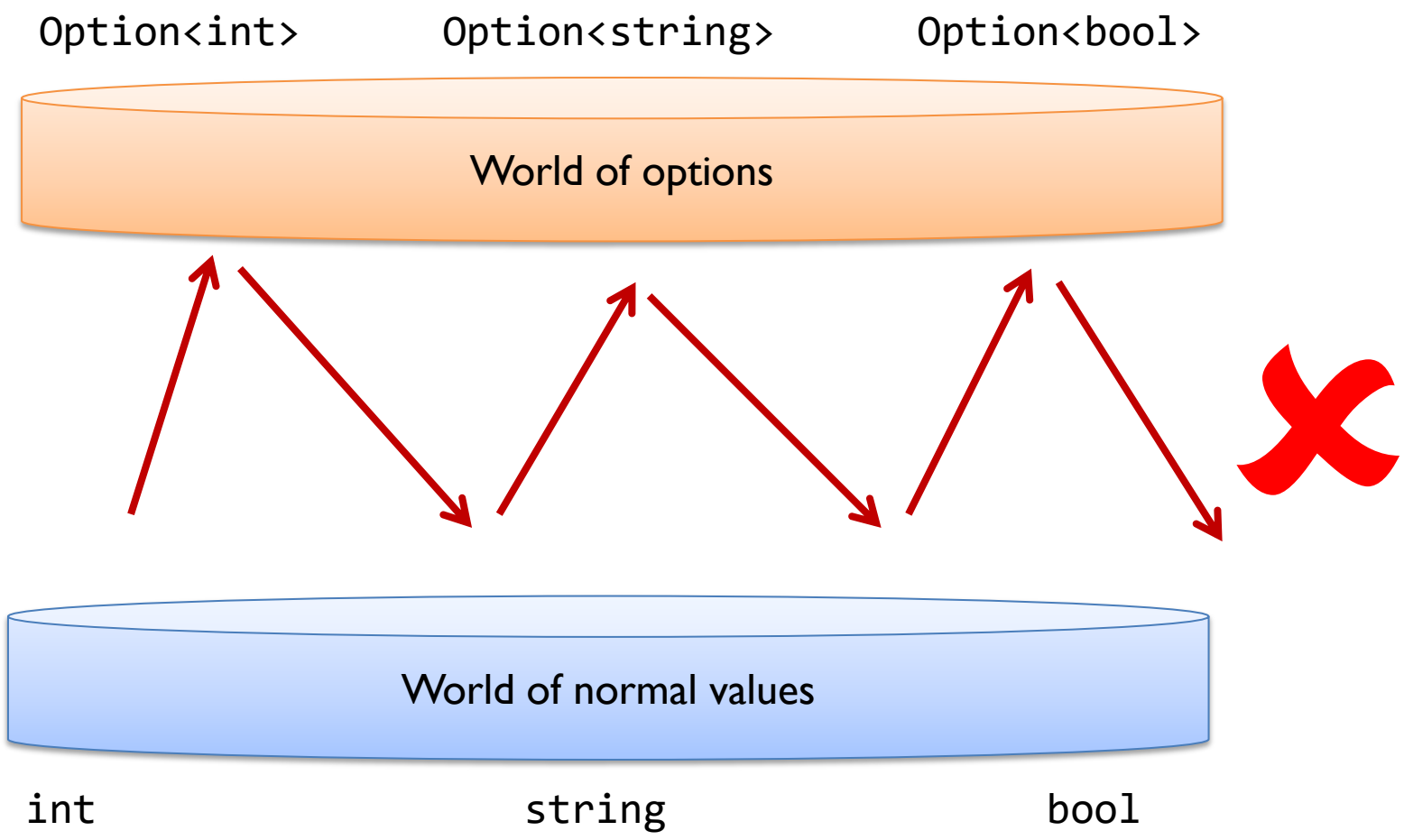
How do we compose these functions together? 

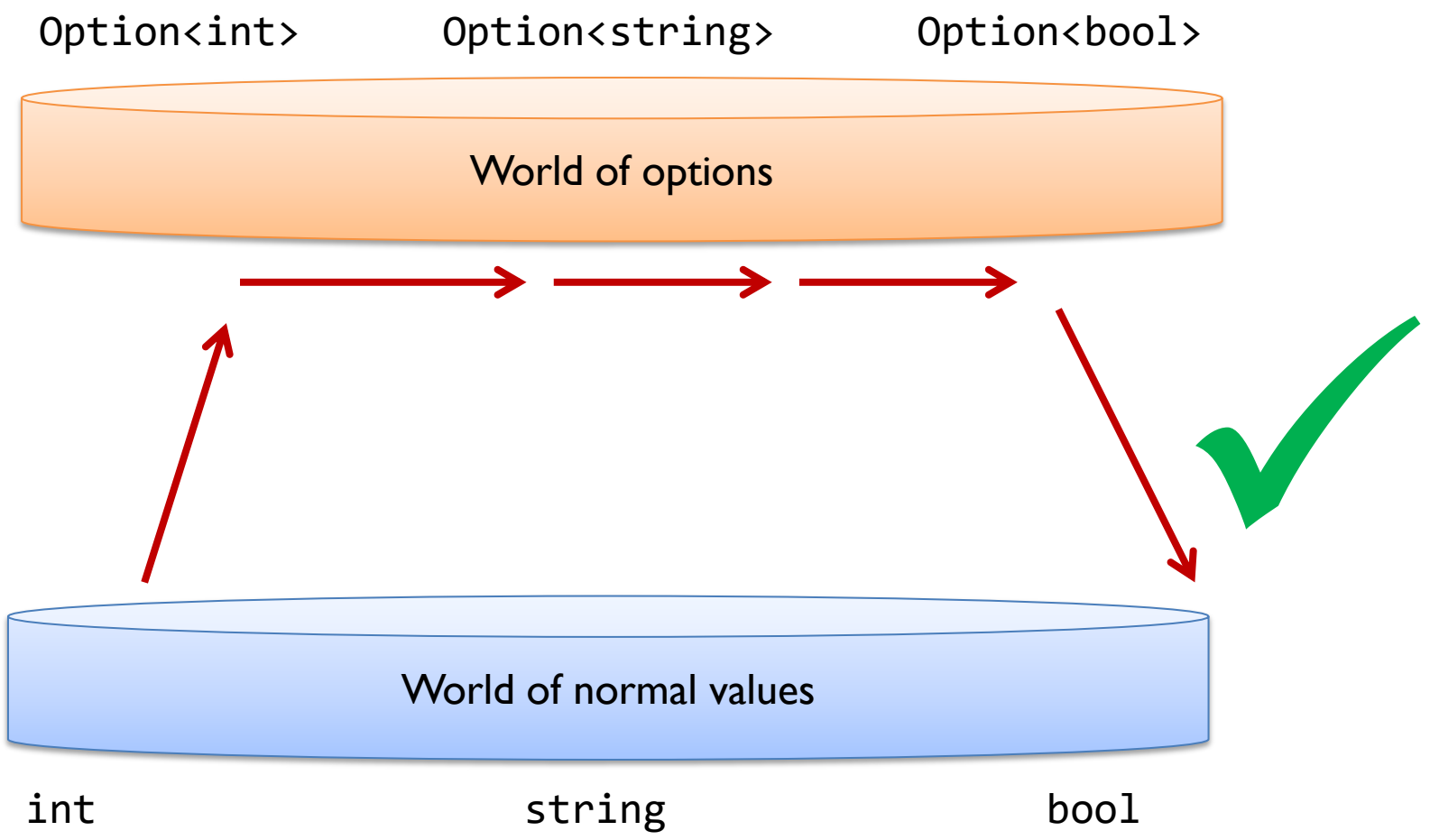
None the worlds match up ...

... but we can use the functional toolkit!

Example:
Working with Options

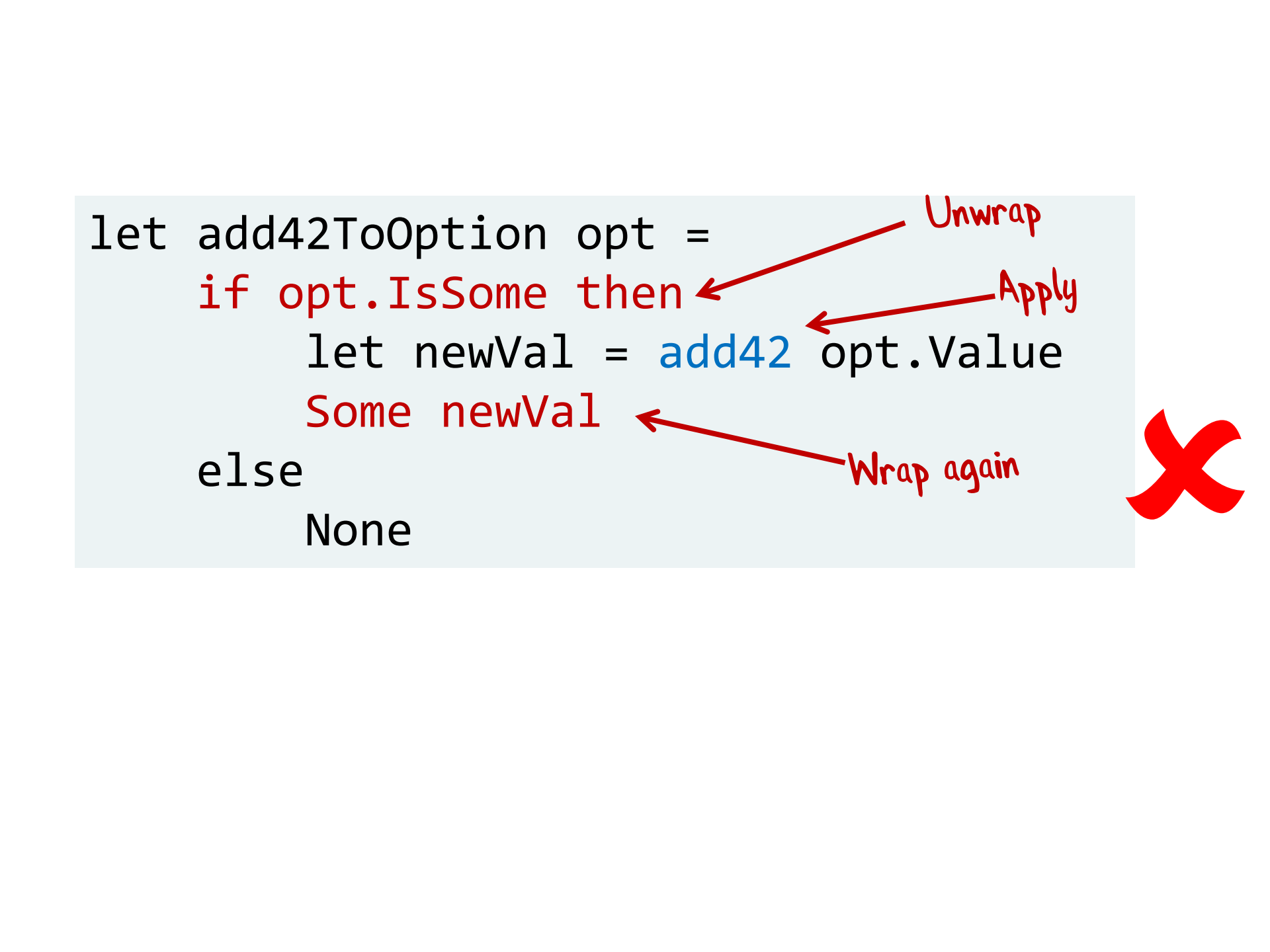

Option<int>
Option<string>
Option<bool>
World of options
World of normal values
int
string
bool

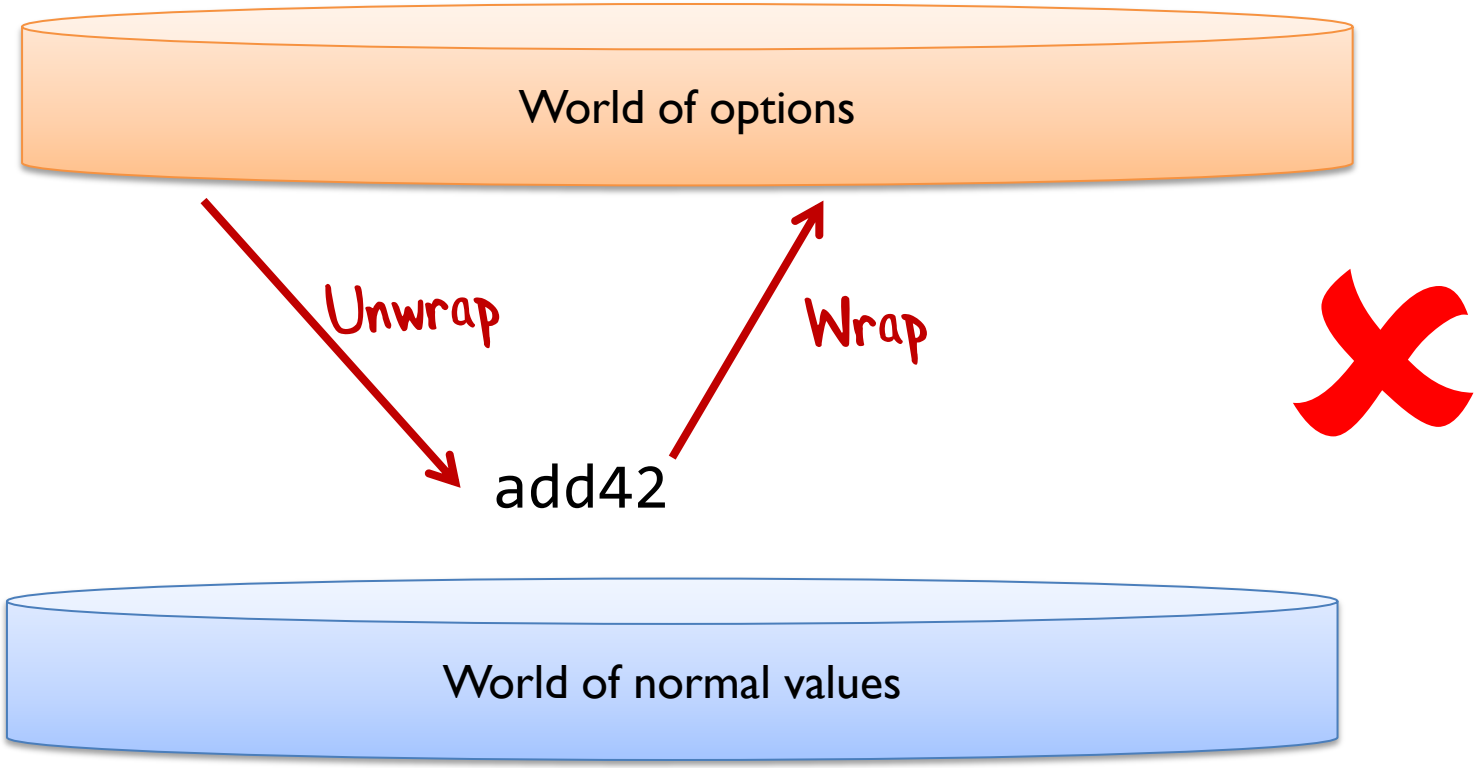

Option<int>
Option<string>
Option<bool>
World of options
World of normal values
int
string
bool

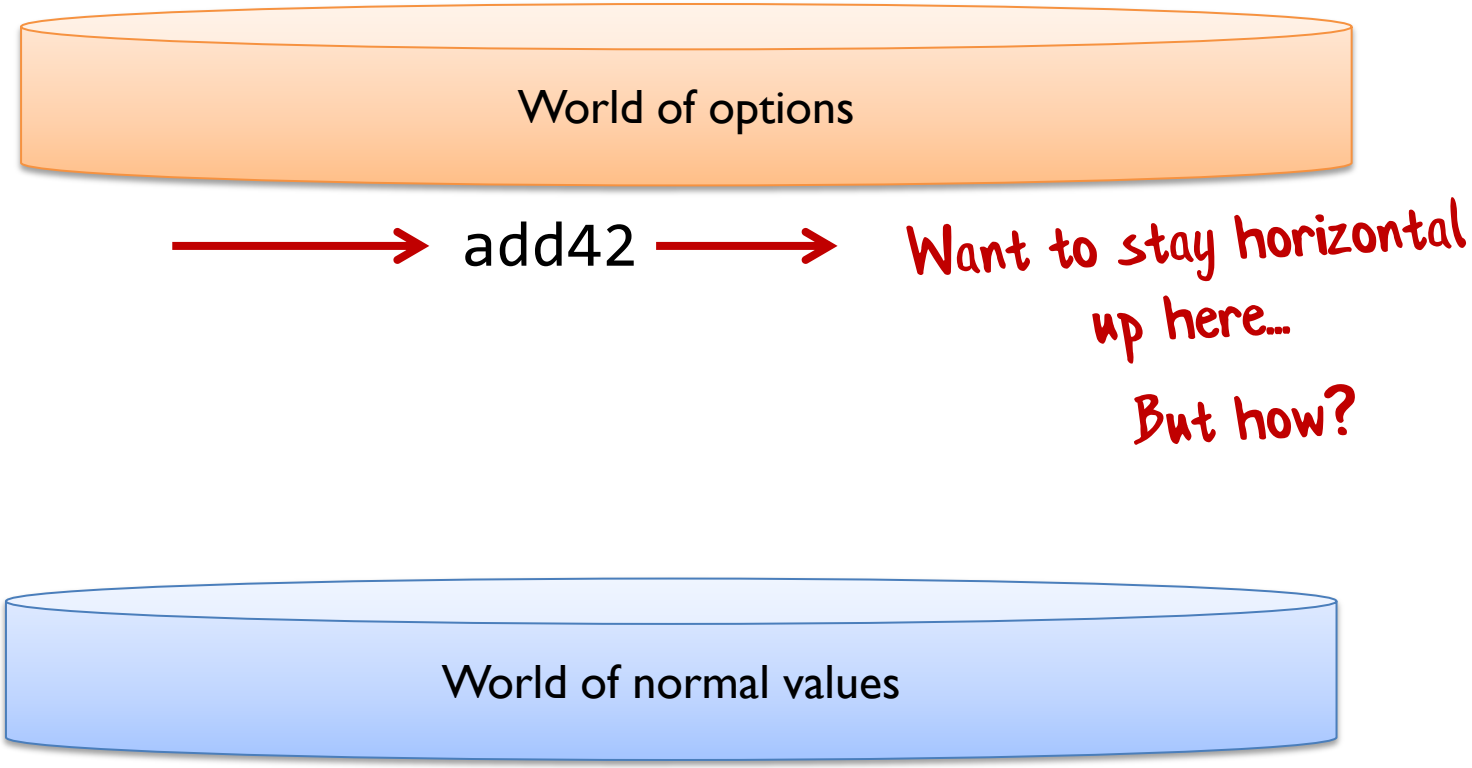

Option<int>
Option<string>
Option<bool>
World of options
World of normal values
int
string
bool

```
let add42 x = x + 42

add42 1   // 43

add42 (Some 1)   // error
```

```
let add42ToOption opt =
    if opt.IsSome then
        let newVal = add42 opt.Value
        Some newVal
    else
        None
```

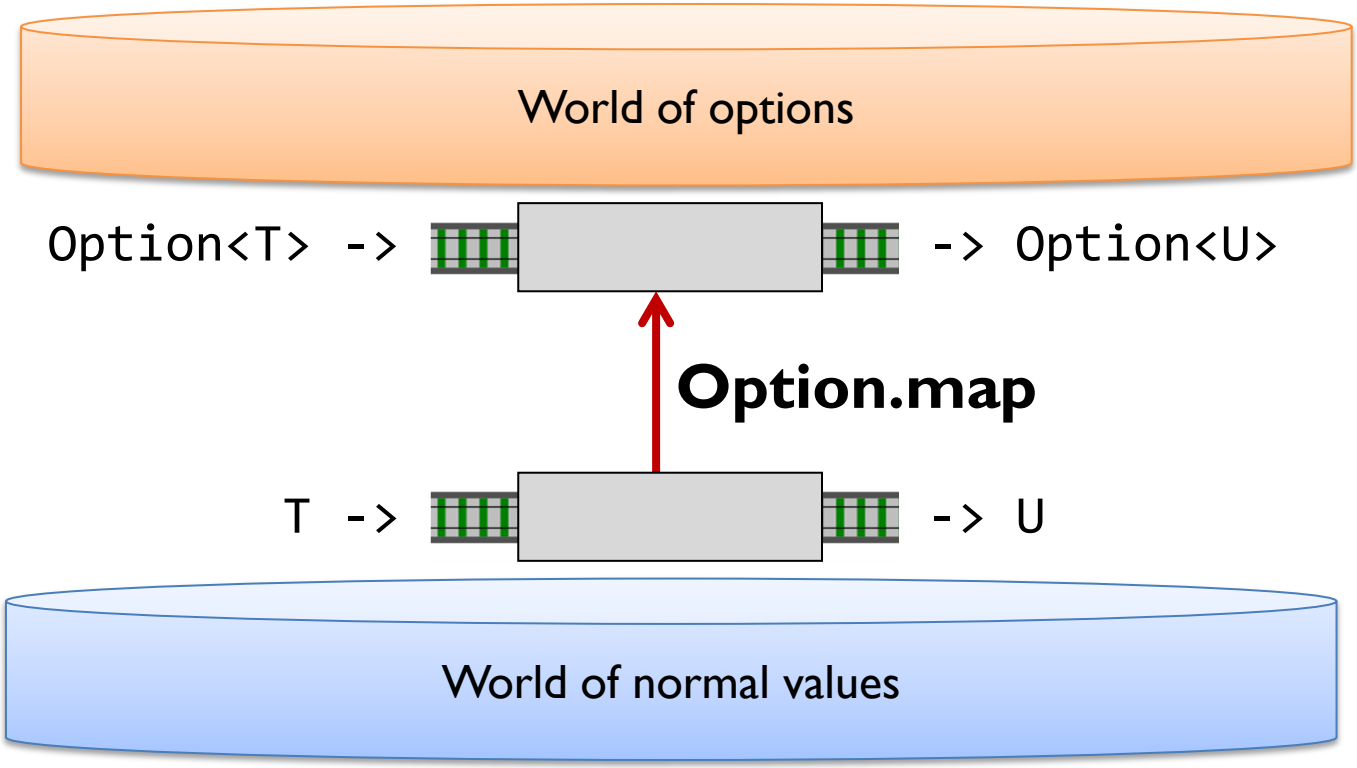



World of options
Unwrap
Wrap
add42
World of normal values


World of options
add42
Want to stay horizontal up here...
But how?
World of normal values

Tool #2

Moving functions between worlds with "map"


World of options
Option<T> ->
-> Option<U>
Option.map
T ->
-> U
World of normal values

A function in normal world

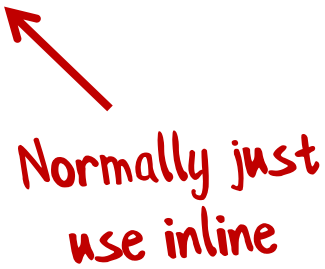
```
let add42 x = ...
```

A function in Option world

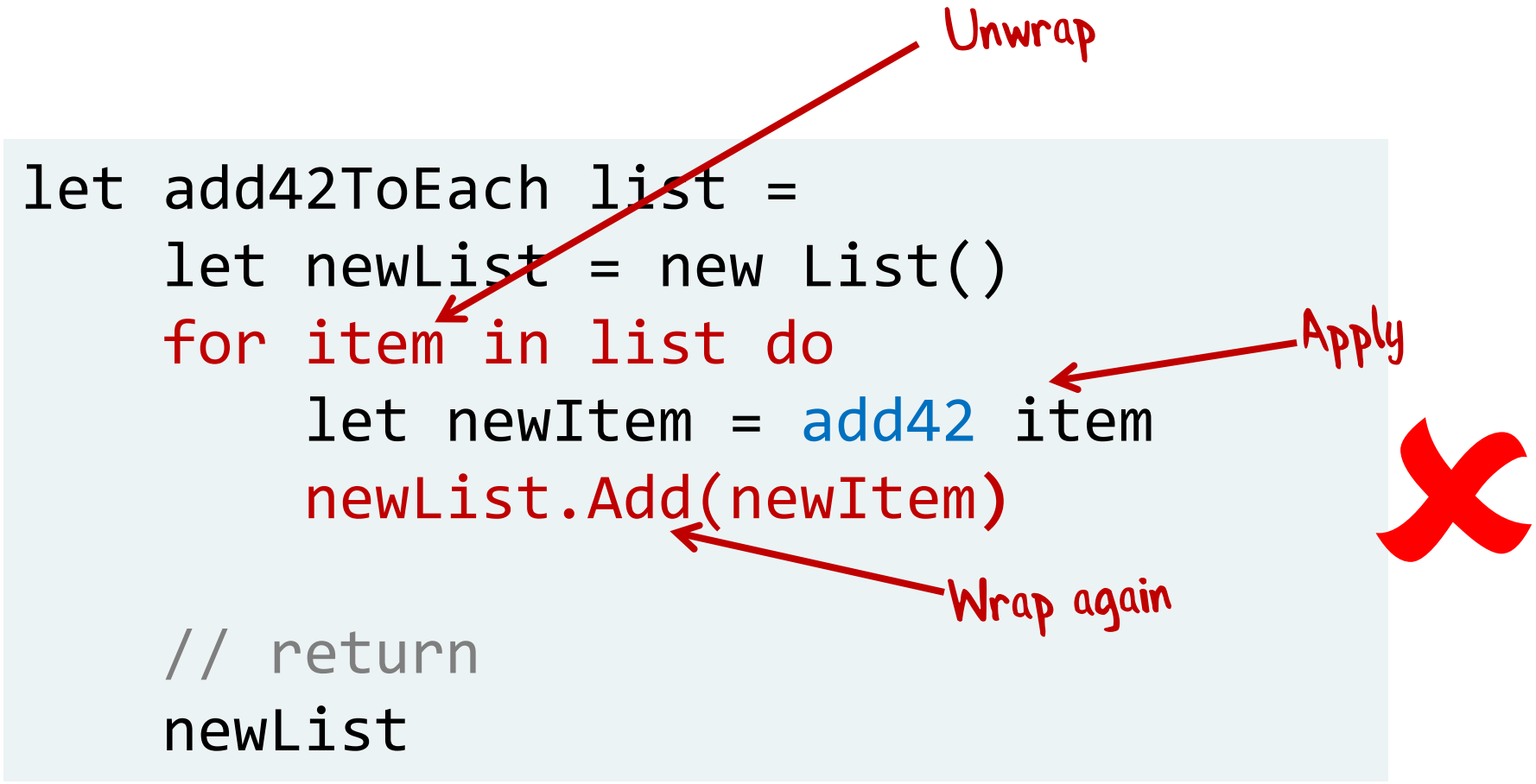
```
let add42ToOption = Option.map add42

add42ToOption (Some 1)  // Some 43
```

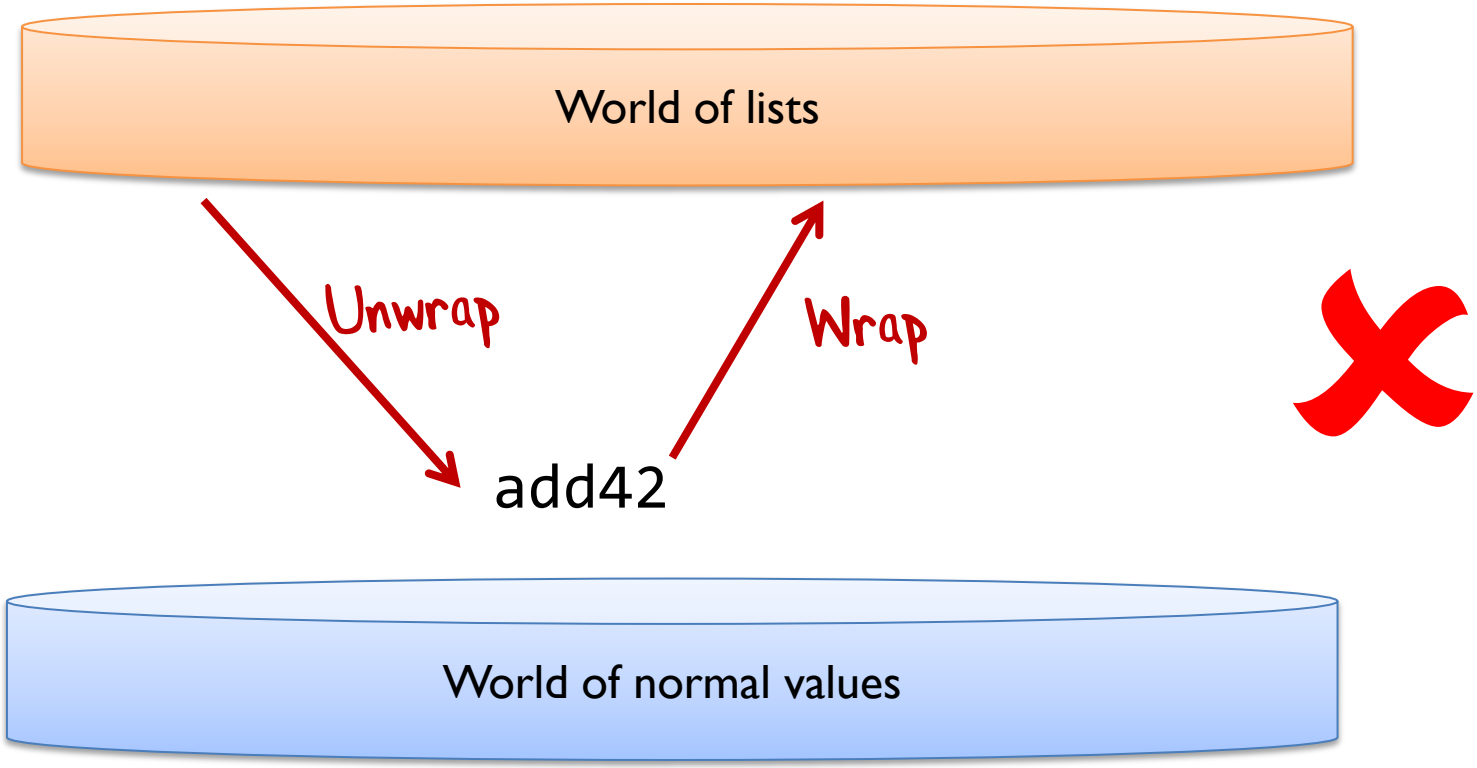
```
let add42 x = ...
```

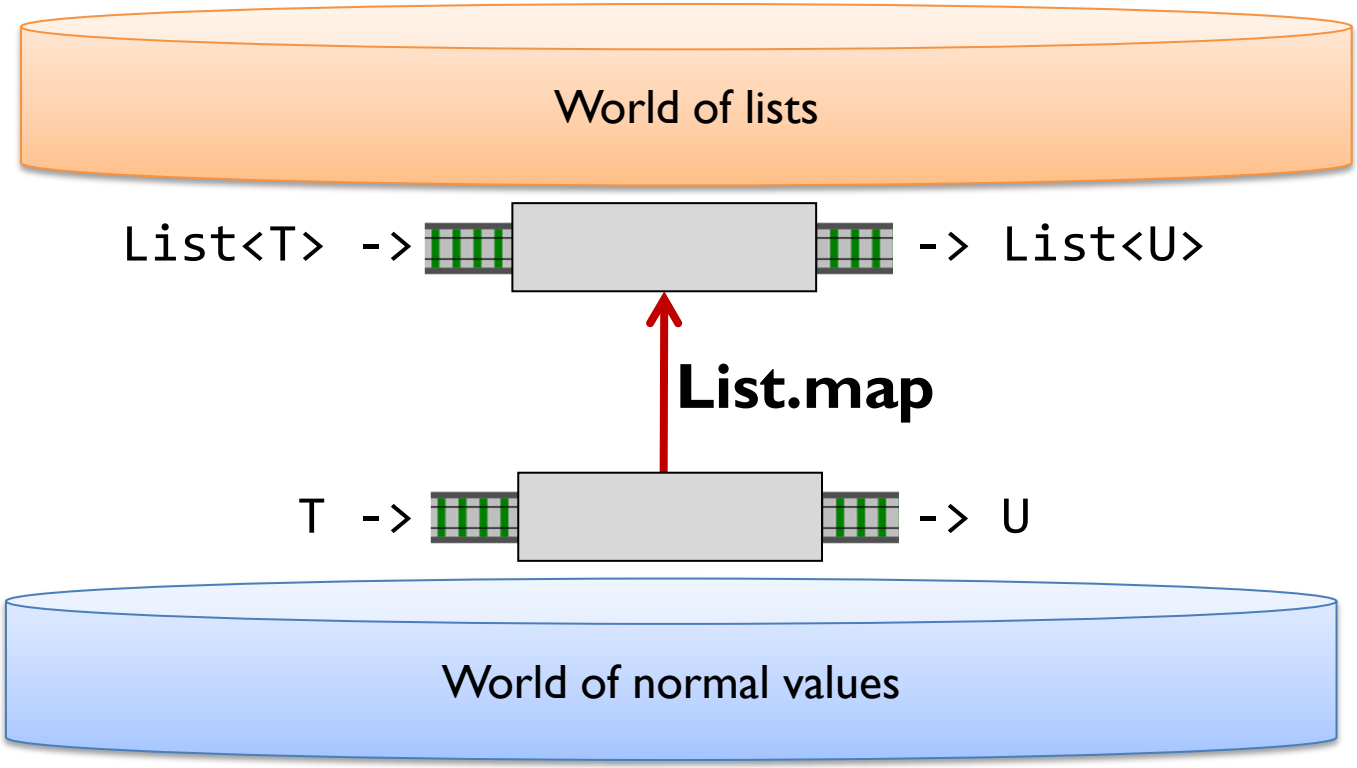
```
(Option.map add42) (Some 1)
```



Example:
Working with List world


Unwrap
let add42ToEach list =
let newList = new List()
for item in list do
let newItem = add42 item
newList.Add(newItem)
// return
newList
Apply
Wrap again


World of lists
Unwrap
Wrap
add42
World of normal values

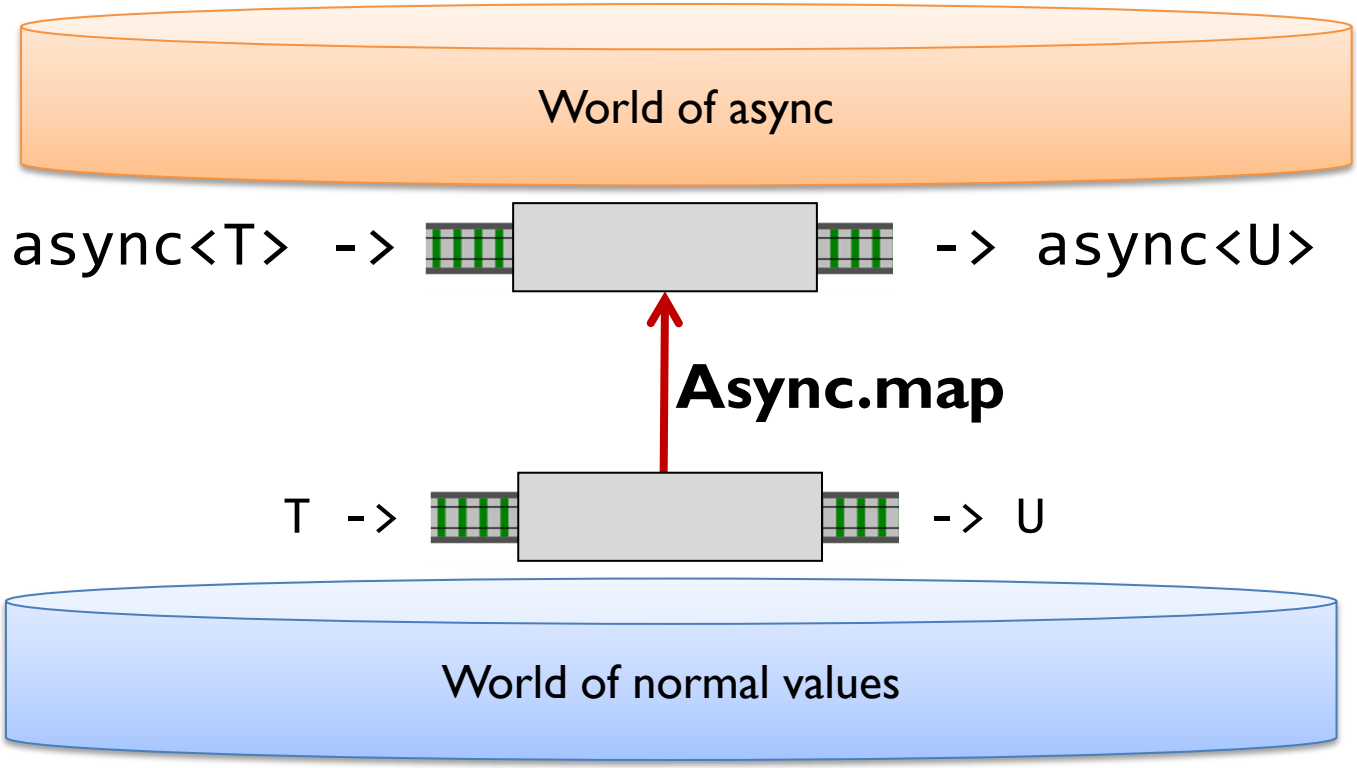

World of lists
List<T> ->
-> List<U>
List.map
T ->
-> U
World of normal values

```
let add42 x = ...
```

A function in List world

```
let add42ToEach = List.map add42

add42ToEach [1;2;3]  // [43;44;45]
```

Is this any better than writing your own loops every time?


World of async
async<T> ->
-> async<U>
Async.map
T ->
-> U
World of normal values

Guideline:

Most wrapped generic types have a “map”. Use it!

Guideline:

If you create your own generic type, create a "map" for it.

FP terminology

A **functor** is

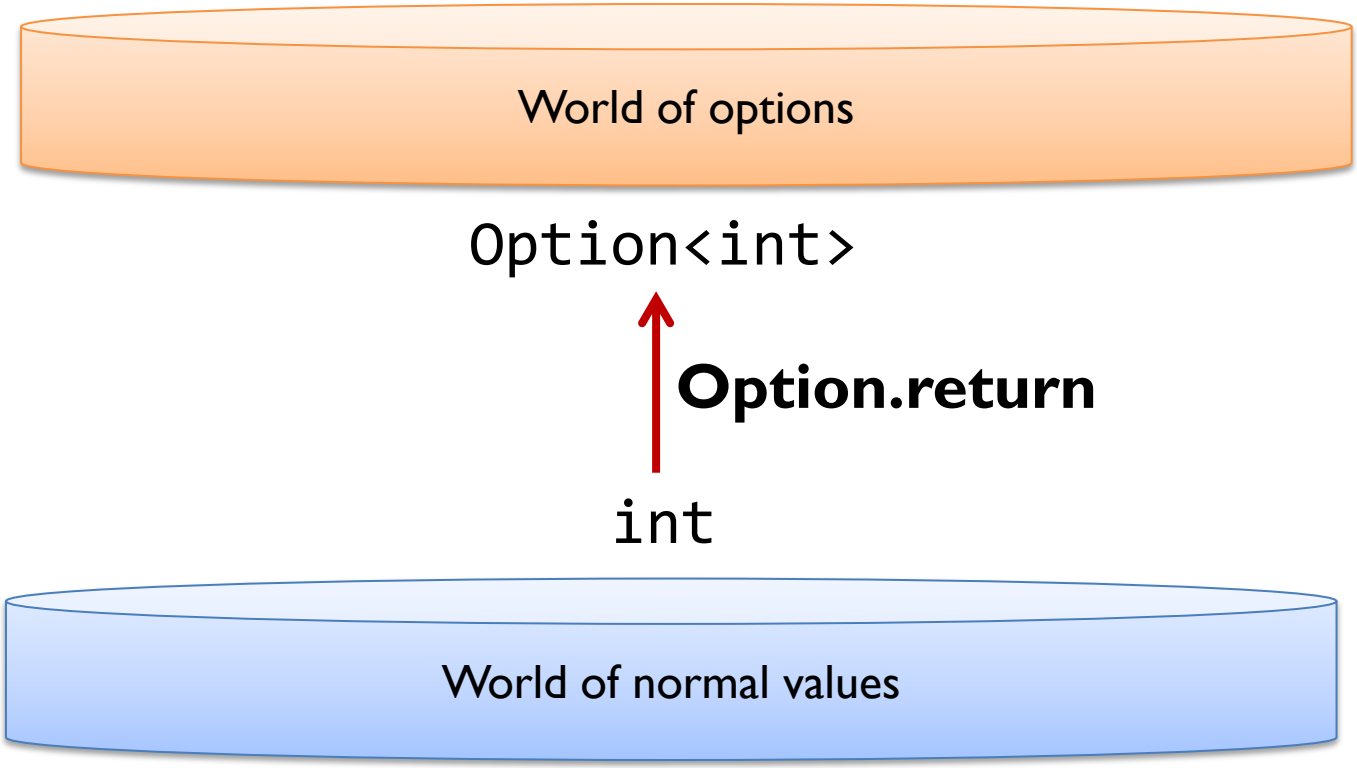
i. An effect type
   - e.g. `Option<>, List<>, Async<>`

ii. Plus a "`map`" function that "lifts" a function to the effects world
   - a.k.a. `select, lift`

iii. And it must have a sensible implementation
   - the Functor laws

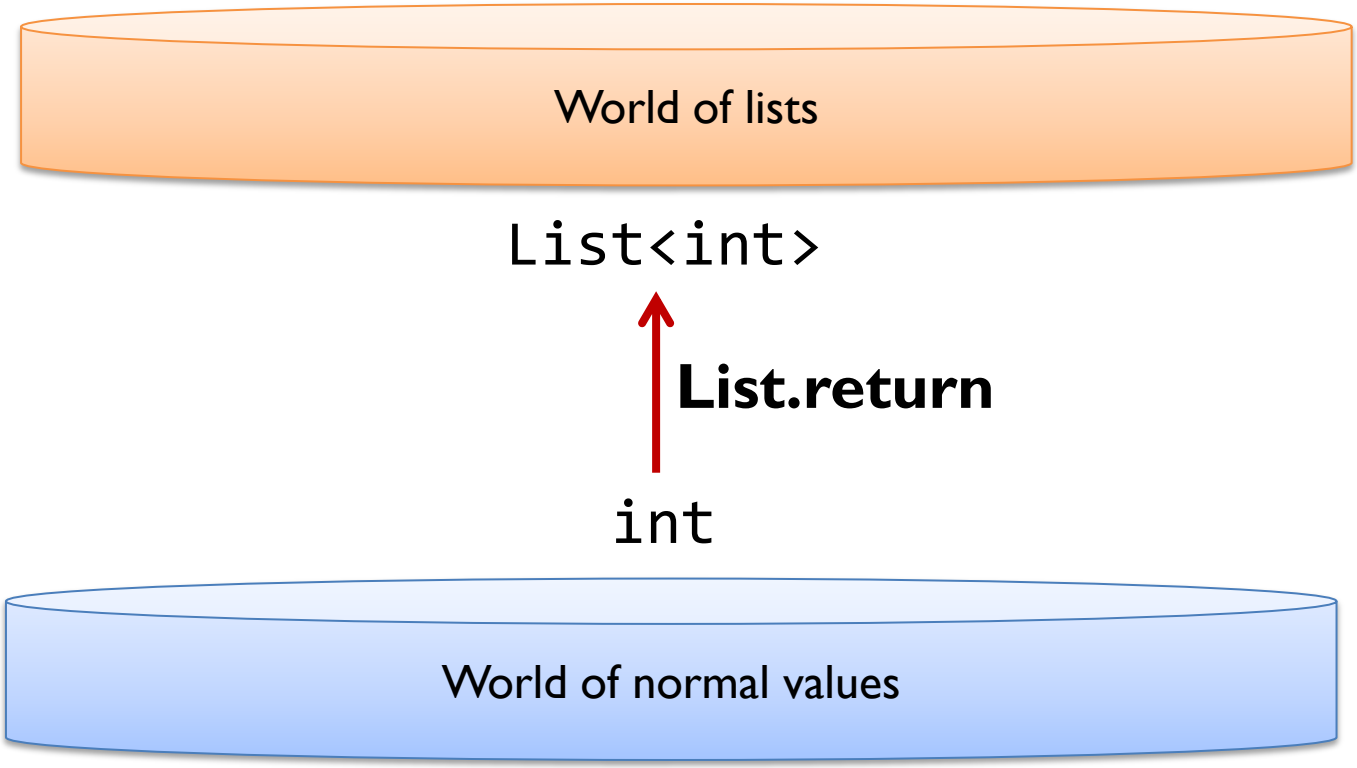
Tool #3

Moving values between worlds with "return"


World of options
Option<int>
Option.return
int
World of normal values


```
let x = 42
```


```
let intOption = Some x
```


World of lists
List<int>
List.return
int
World of normal values


```
let x = 42
```


```
let intList = [x]
```

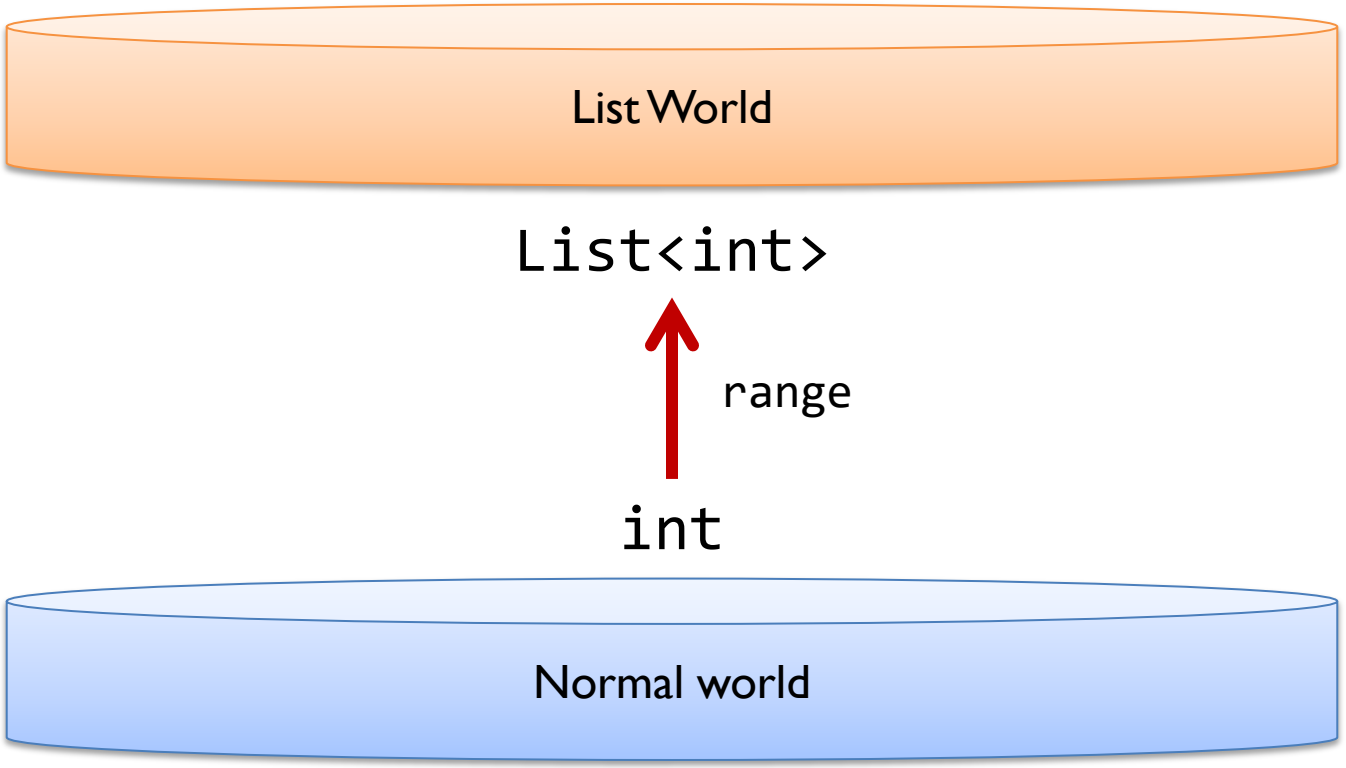
Tool #4

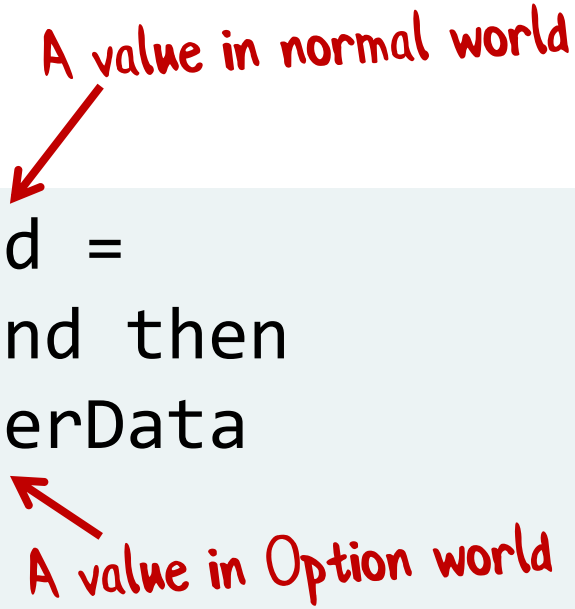
Chaining world-crossing functions with "bind"

What's a world-crossing function?


```
let range max = [1..max]

// int -> List<int>
```

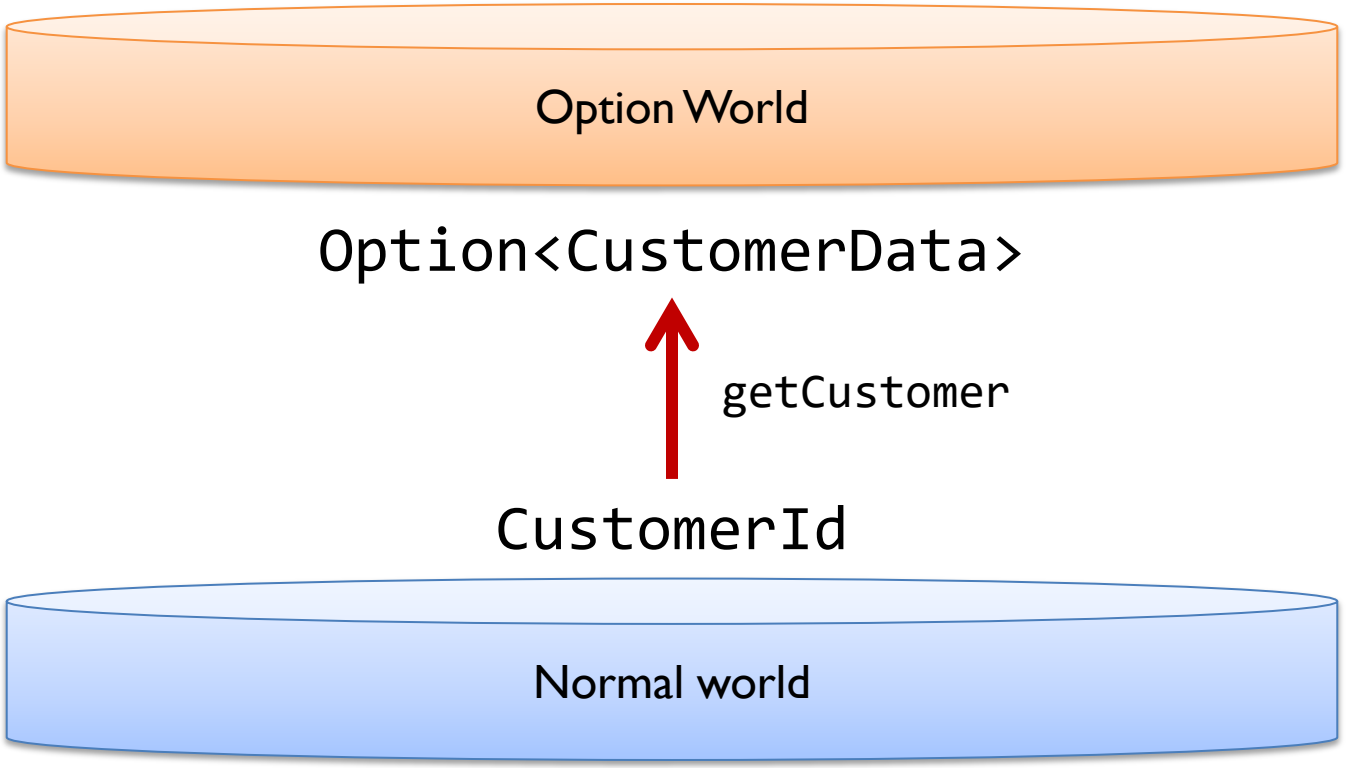
A world crossing function



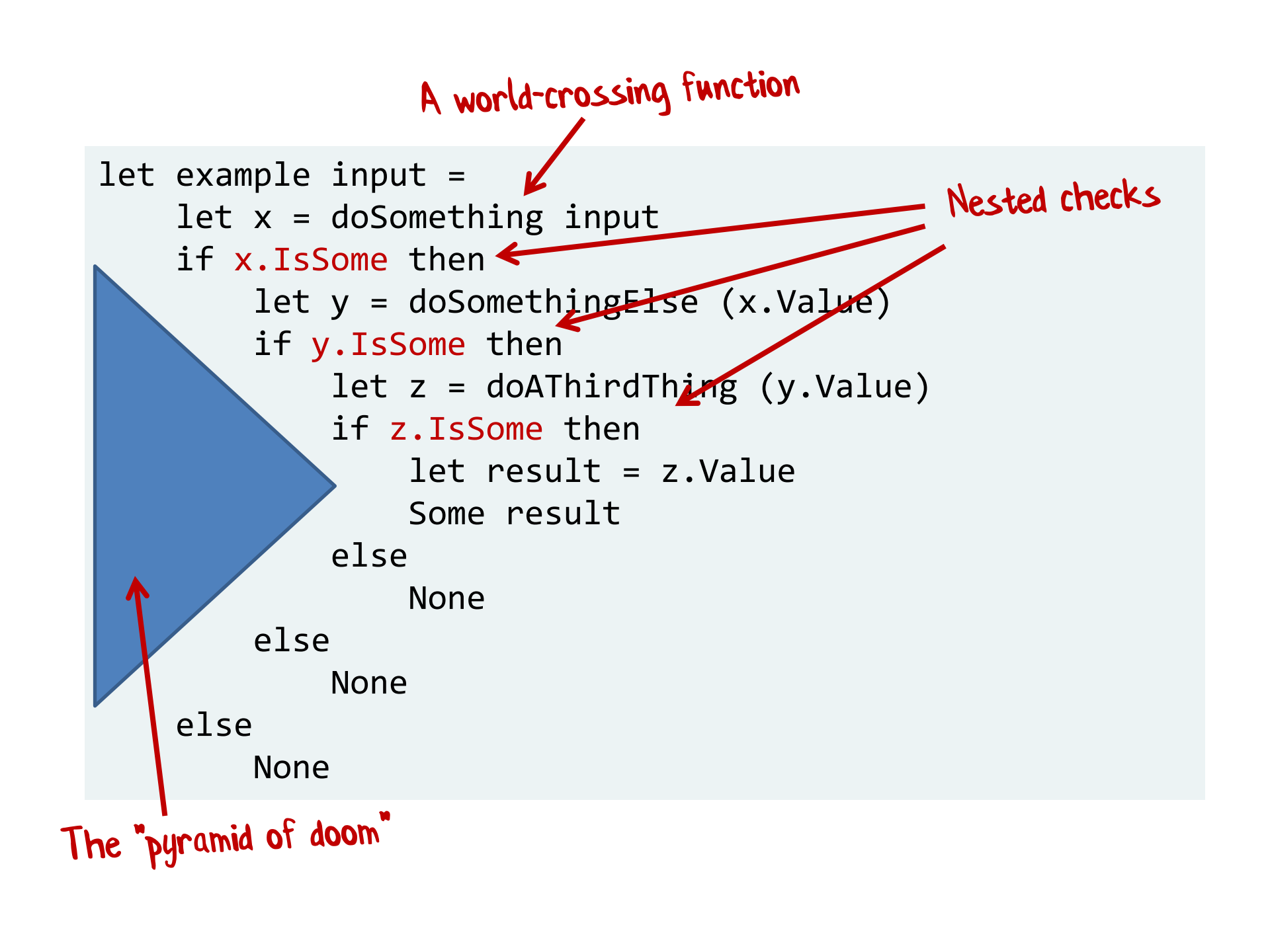



```
let getCustomer id =
    if customerFound then
        Some customerData
    else
        None

// CustomerId -> Option<CustomerData>
```

A world crossing function



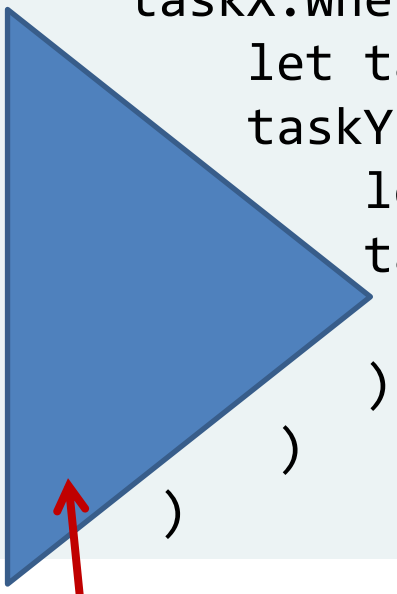
Problem:
How do you chain
world-crossing functions?


A world-crossing function
Nested checks
let example input =
let x = doSomething input
if x.IsSome then
let y = doSomethingElse (x.Value)
if y.IsSome then
let z = doAThirdThing (y.Value)
if z.IsSome then
let result = z.Value
Some result
else
None
else
None
else
None
The "pyramid of doom"

A world-crossing function

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                z  // final result
            )
        )
    )
```

Nested callbacks



Another "pyramid of doom"

Let's fix this!

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                // do something with z.Value
                // in this block
            else
                None
        else
            None
    else
        None
```

There is a pattern we can exploit...

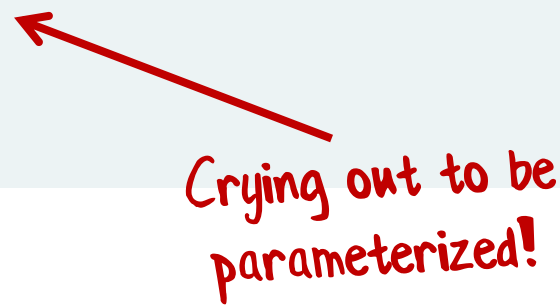
```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                // do something with z.Value
                // in this block
            else
                None
        else
            None
    else
        None
```

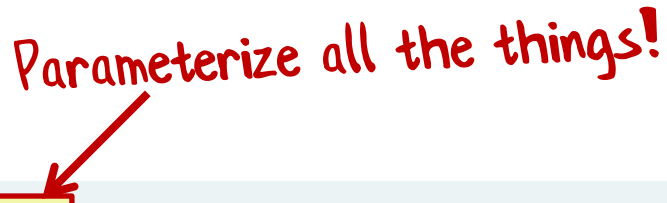
```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            // do something with y.Value
            // in this block



        else
            None
    else
        None
```

```
let example input =
    let x = doSomething input
    if x.IsSome then
        // do something with x.Value
        // in this block




    else
        None
```

Can you see the pattern?

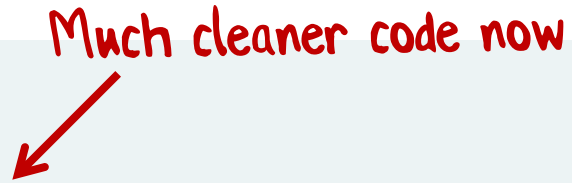
```
if opt.IsSome then
    //do something with opt.Value
else
    None
```





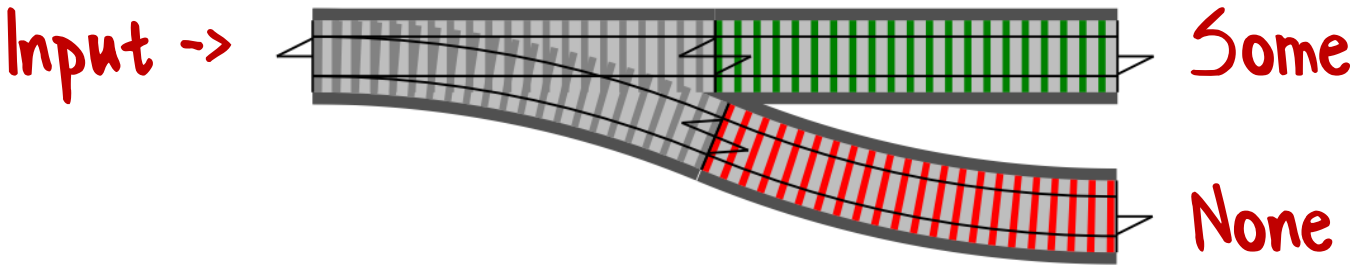

```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```

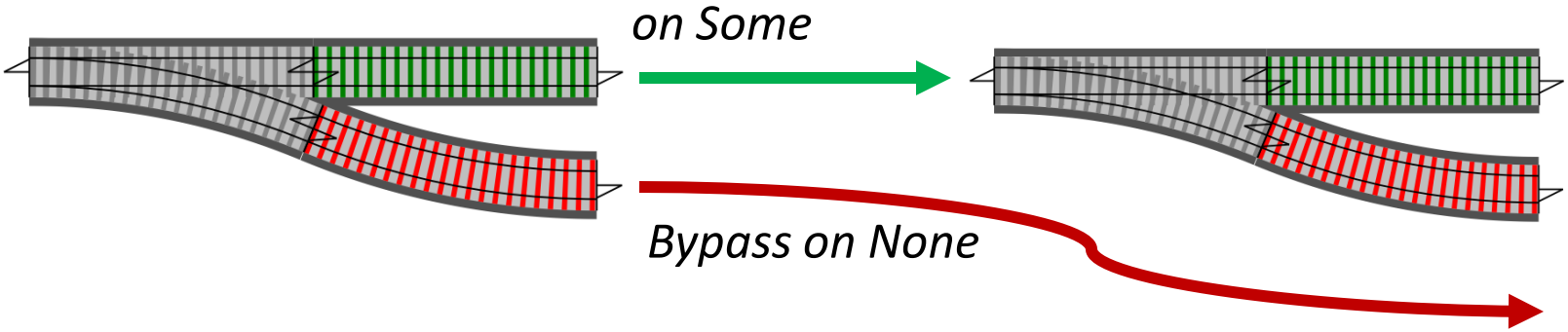
```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```

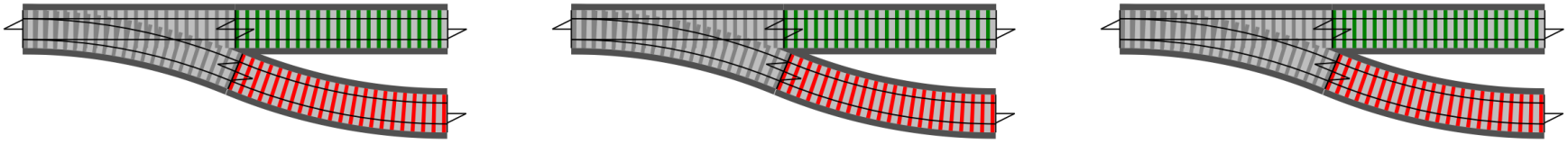


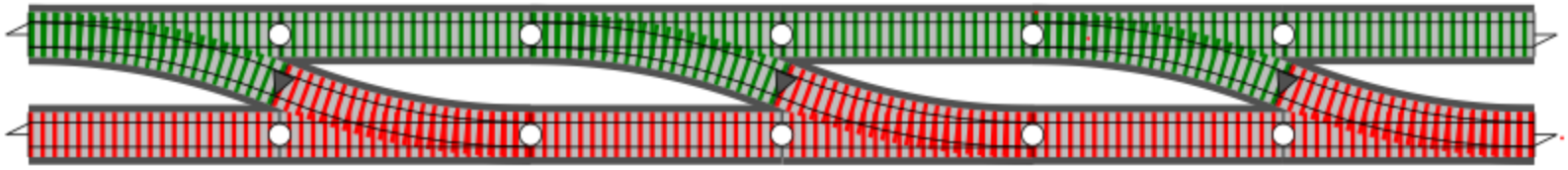

```
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo ...
```

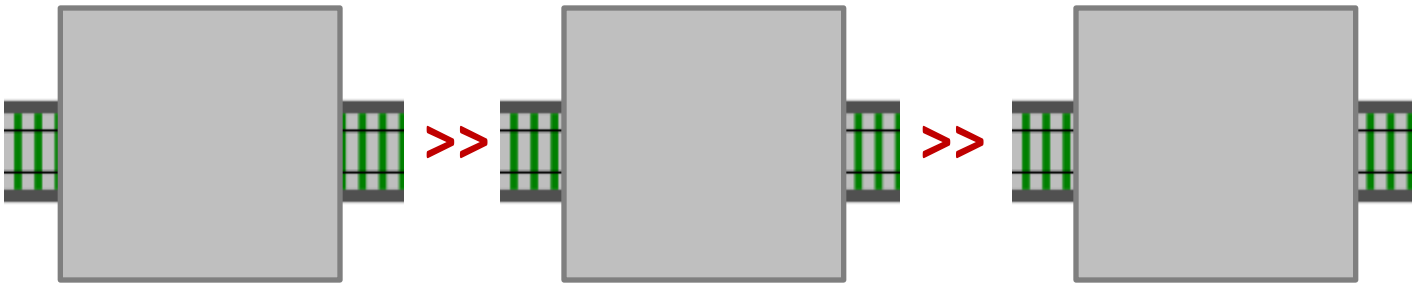
Let's use a railway analogy



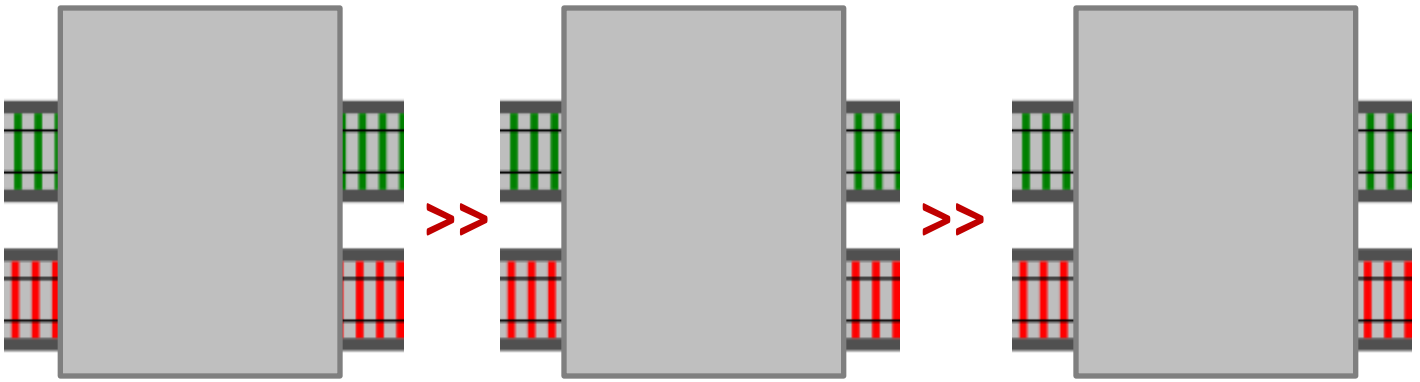

on Some
Bypass on None



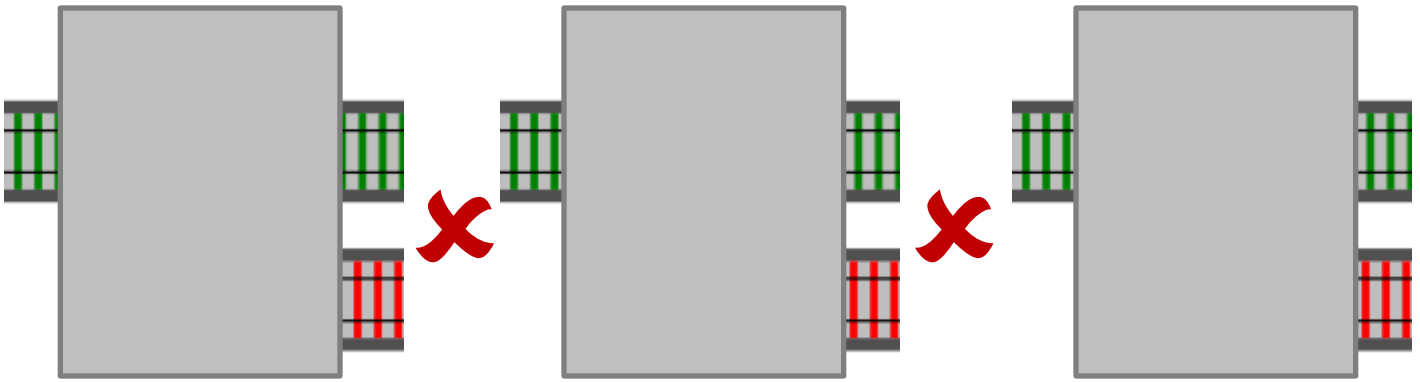




Composing one-track functions is fine...



... and composing two-track functions is fine...

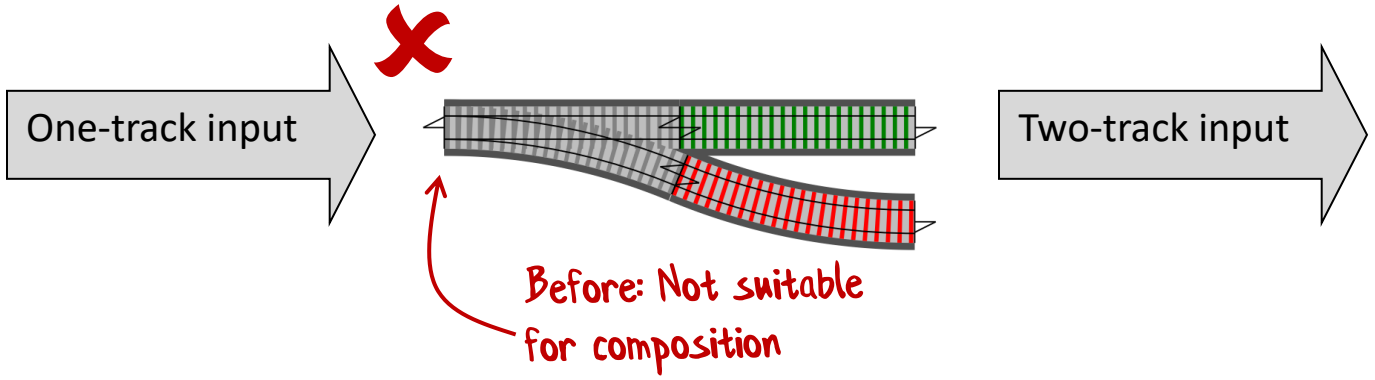


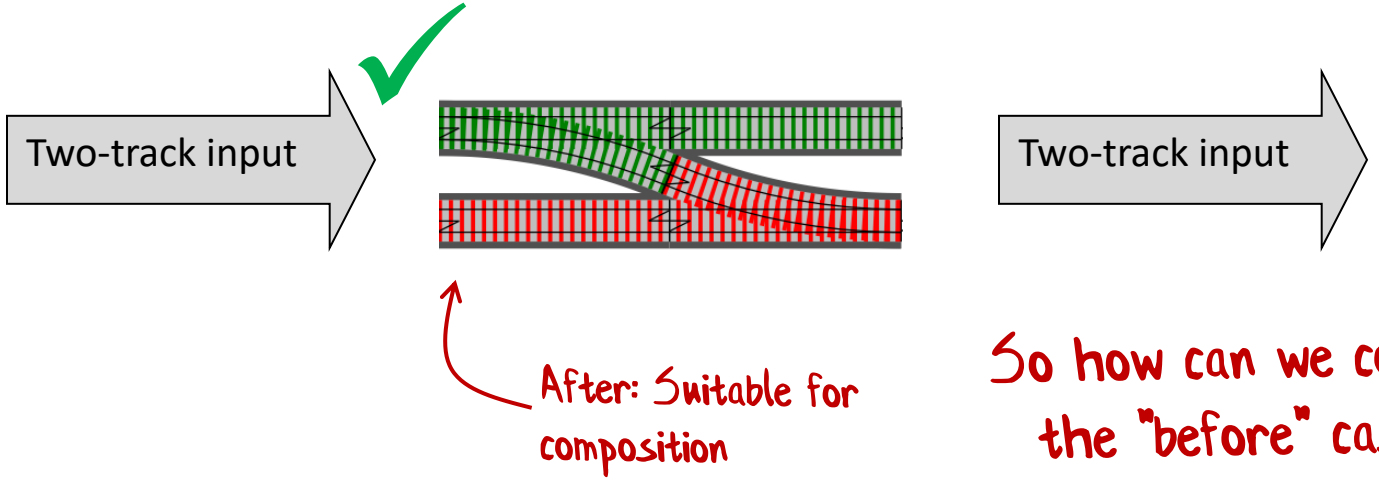
... but composing switches is not allowed!

How to combine the mismatched functions?

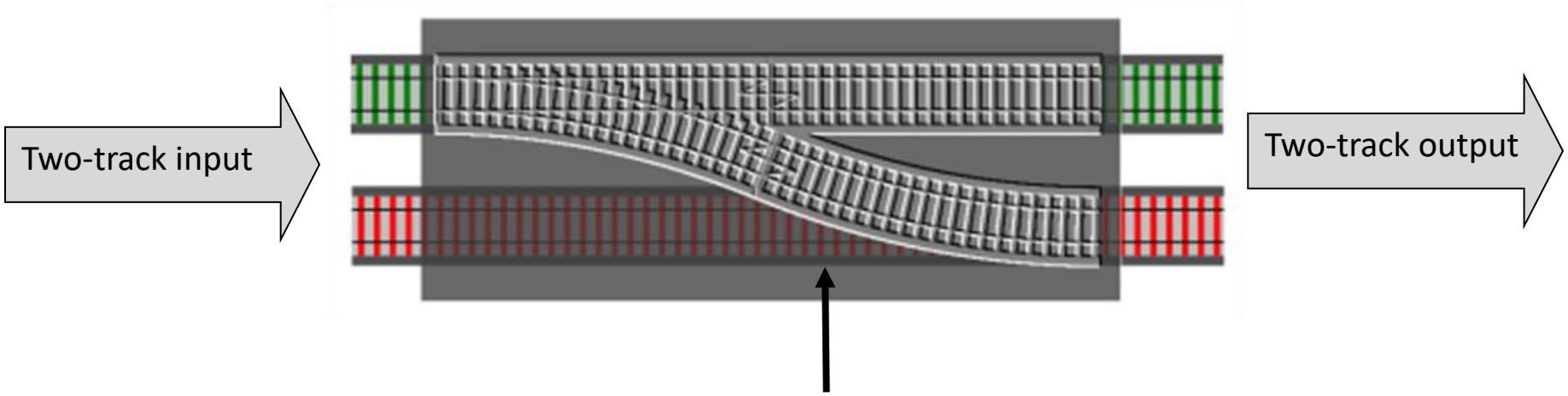
"Bind" is the answer!
Bind all the things!

FP'ers get excited by bind

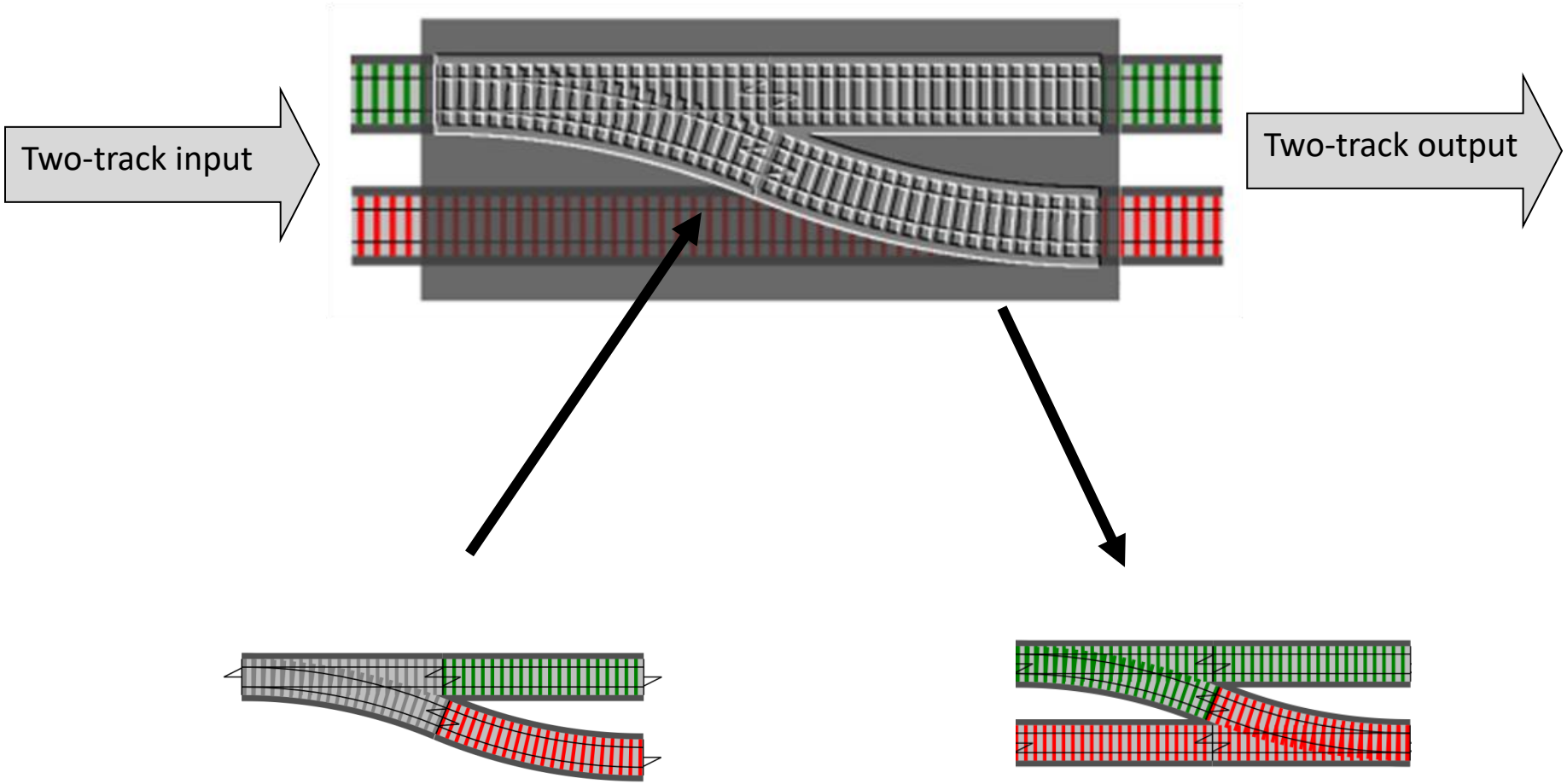


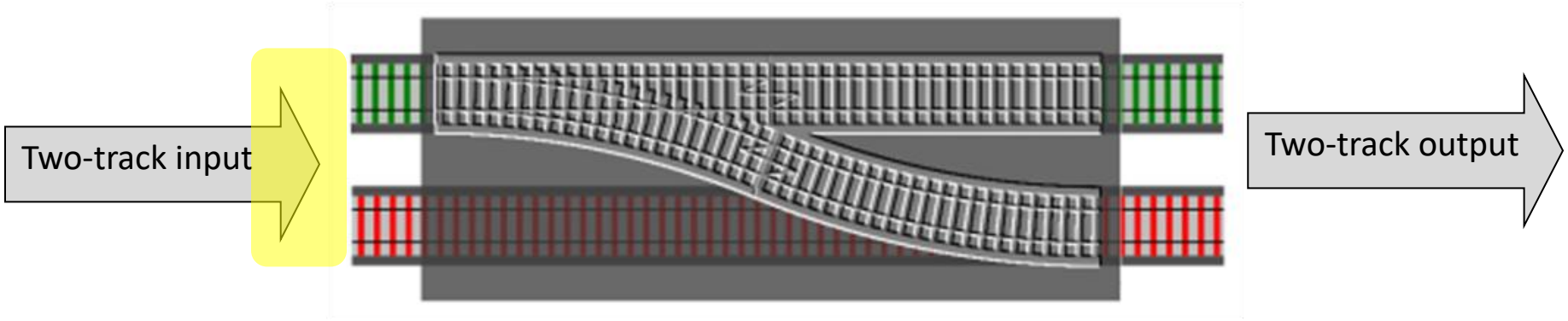


So how can we convert from the "before" case to the "after" case?

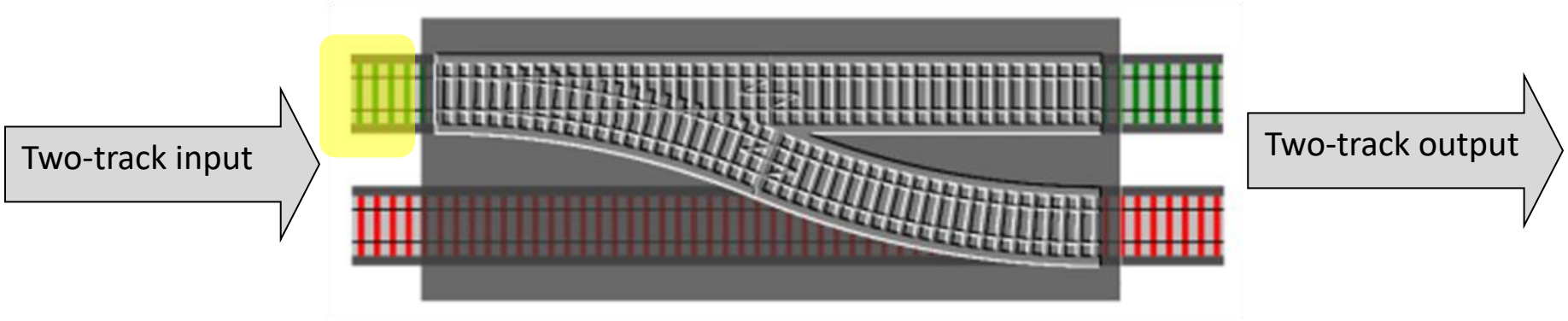

Two-track input
Two-track output
Slot for switch function

A "function transformer"

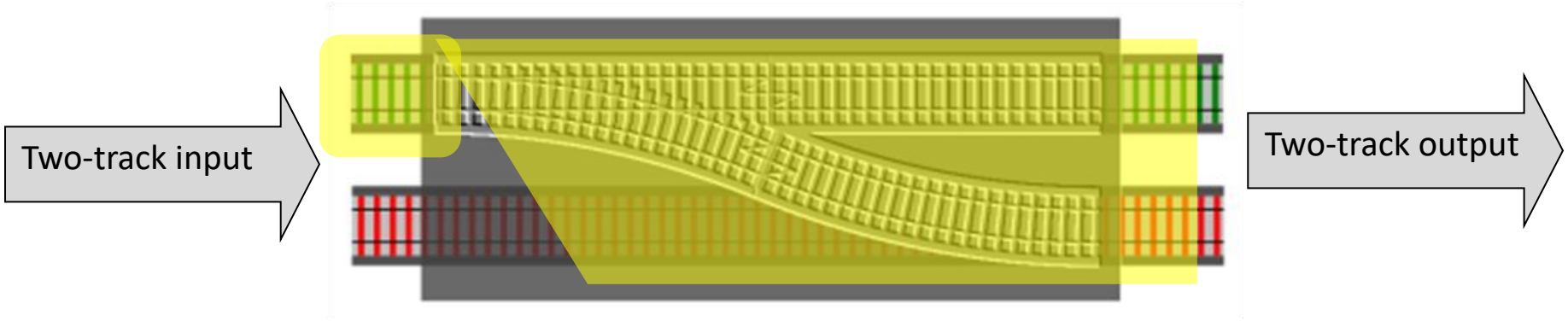




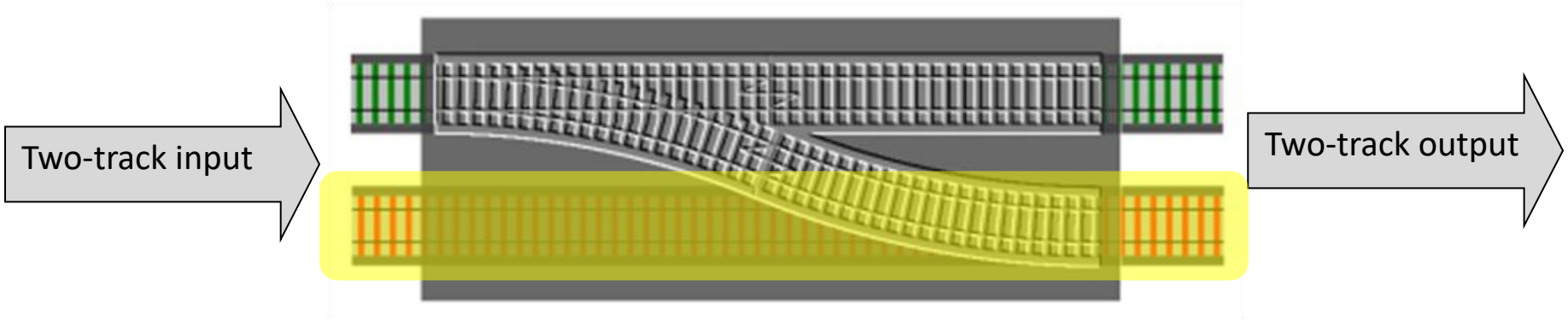

```
let bind nextFunction optionInput =
    match optionInput with
    | Some s -> nextFunction s
    | None -> None
```




```
let bind nextFunction optionInput =
    match optionInput with
    | Some s -> nextFunction s
    | None -> None
```




```
let bind nextFunction optionInput =
    match optionInput with
    | Some s -> nextFunction s
    | None -> None
```




```
let bind nextFunction optionInput =
    match optionInput with
    | Some s -> nextFunction s
    | None -> None
```

Pattern:
Use bind to chain options

Before

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                let result = z.Value
                Some result
            else
                None
        else
            None
    else
        None
```

After

```
let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

Same as "ifSomeDo"

After

```
let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

```
let example input =
    doSomething input
    |> bind doSomethingElse
    |> bind doAThirdThing
    |> bind ...
```

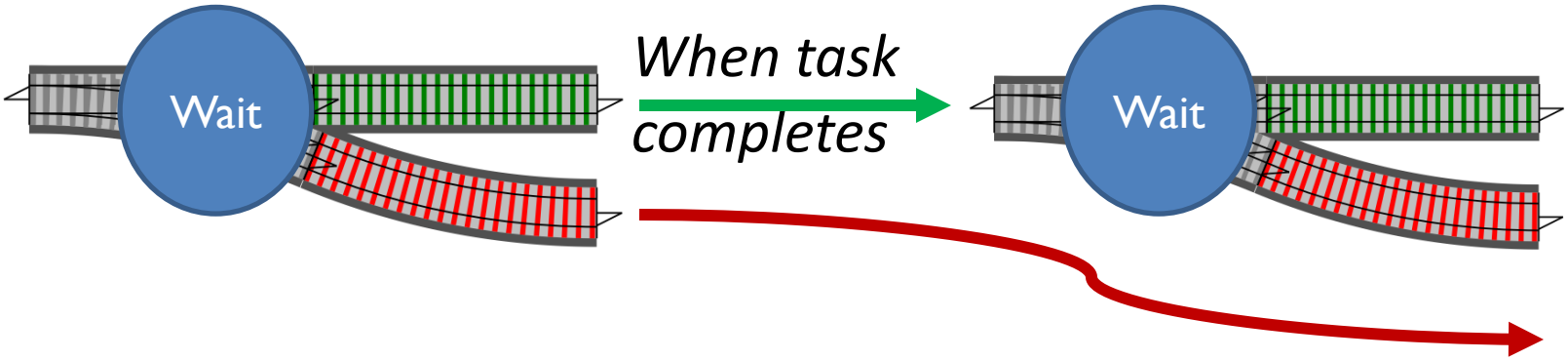
No pyramids!

Code is linear and clear.

Pattern:

Use bind to chain tasks

a.k.a "promise" "future"


Wait
When task
completes
Wait

Before

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                z  // final result
            )
        )
    )
```

After

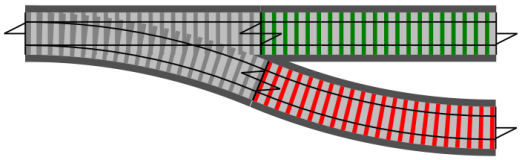
```
let taskBind f task =
    task.WhenFinished (fun taskResult ->
        f taskResult)
```

```
let taskExample input =
    startTask input
    |> taskBind startAnotherTask
    |> taskBind startThirdTask
    |> taskBind ...
```

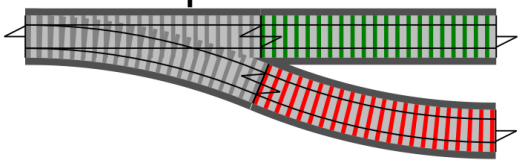
Problem:

How to handle errors elegantly?

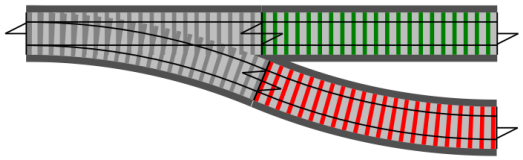
Validate

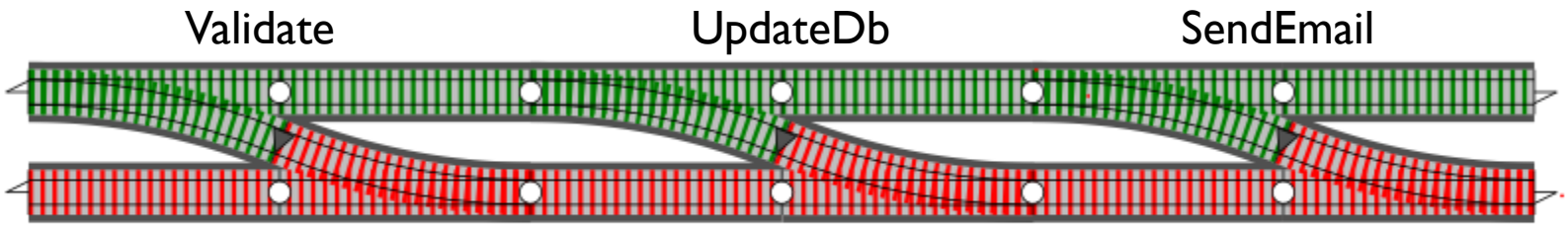


UpdateDb



SendEmail





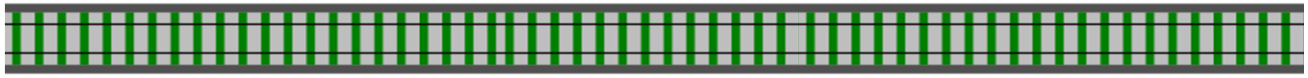
This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

Before

```
let updateCustomer =
  receiveRequest()
  |> validateRequest
  |> canonicalizeEmail
  |> updateDbFromRequest
  |> sendEmail
  |> returnMessage
```

Functional flow without error handling

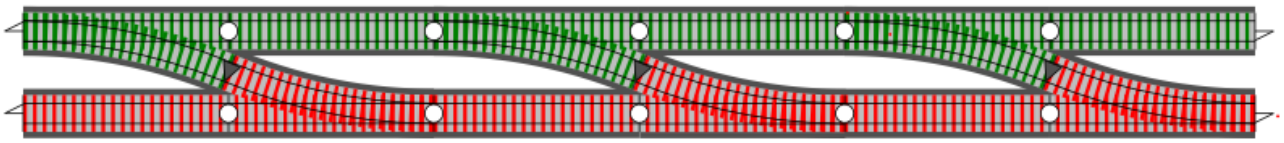
One track



After

```
let updateCustomerWithErrorHandling =
  receiveRequest()
  |> validateRequest
  |> canonicalizeEmail
  |> updateDbFromRequest
  |> sendEmail
  |> returnMessage
```

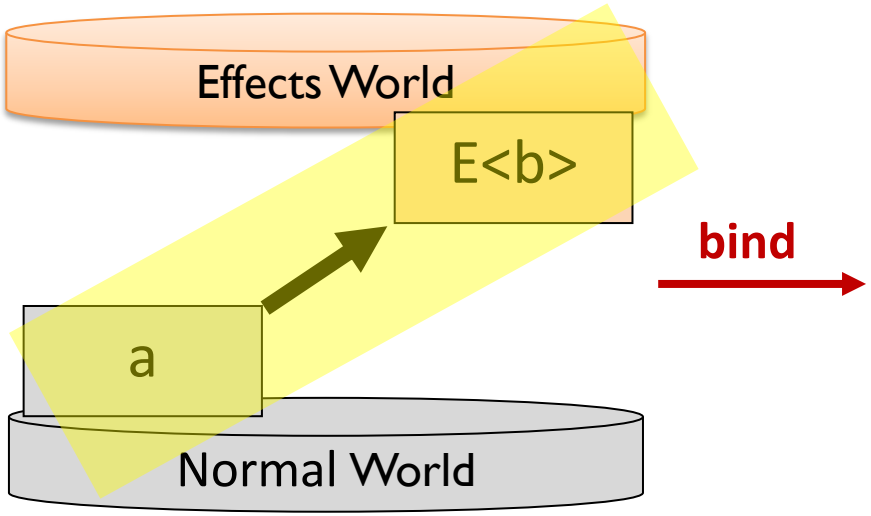
Two track



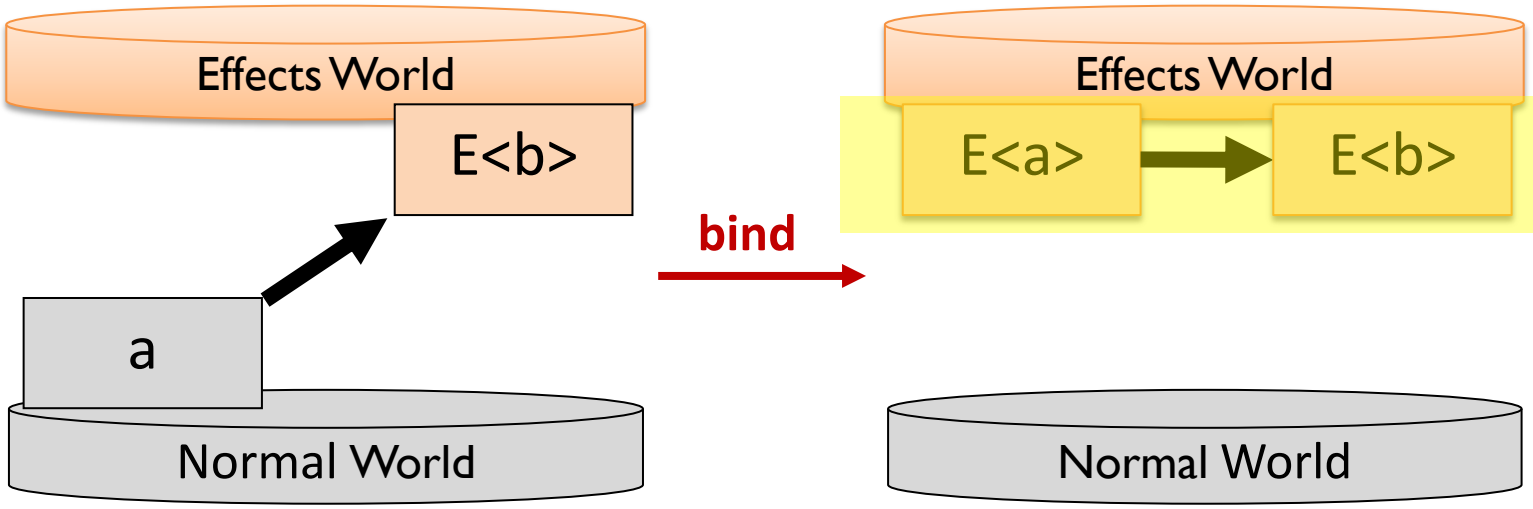
See fsharpforfunandprofit.com/rop

Why is bind so important?

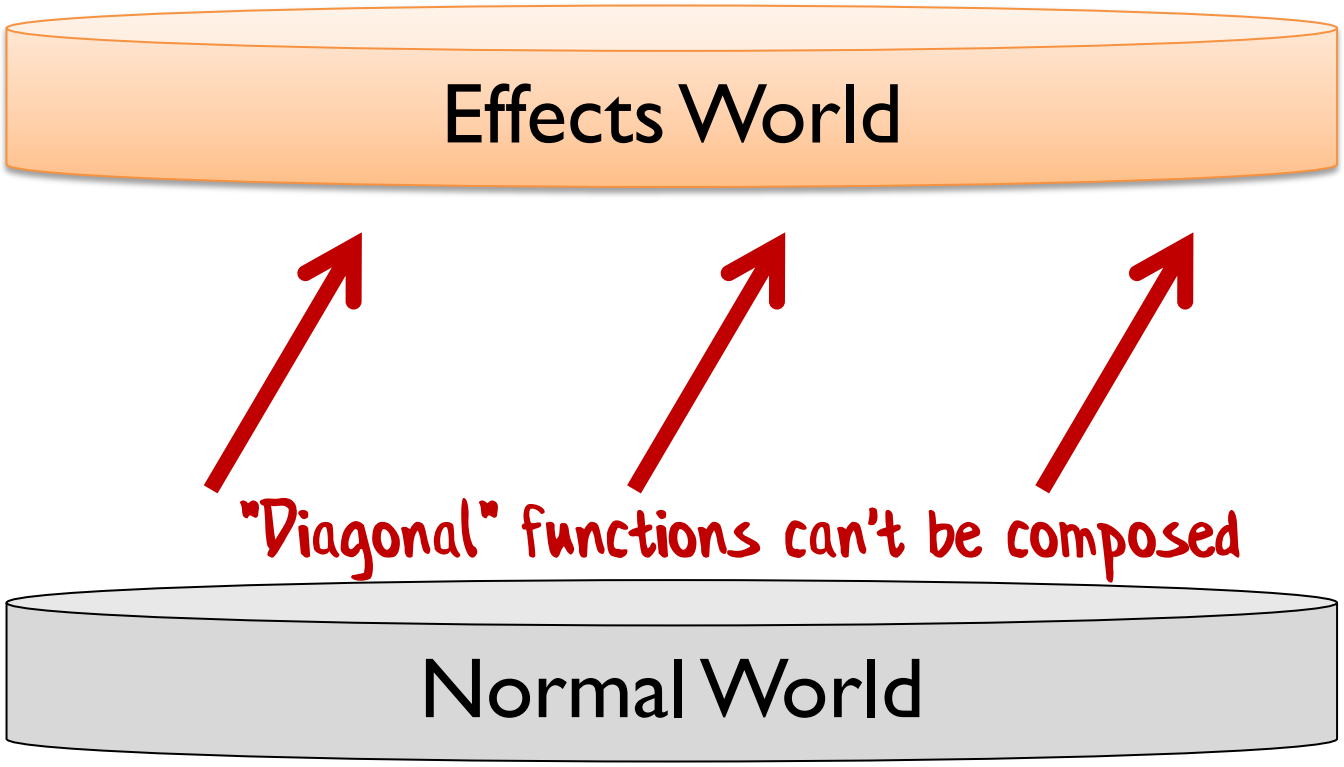
It makes world-crossing functions composable

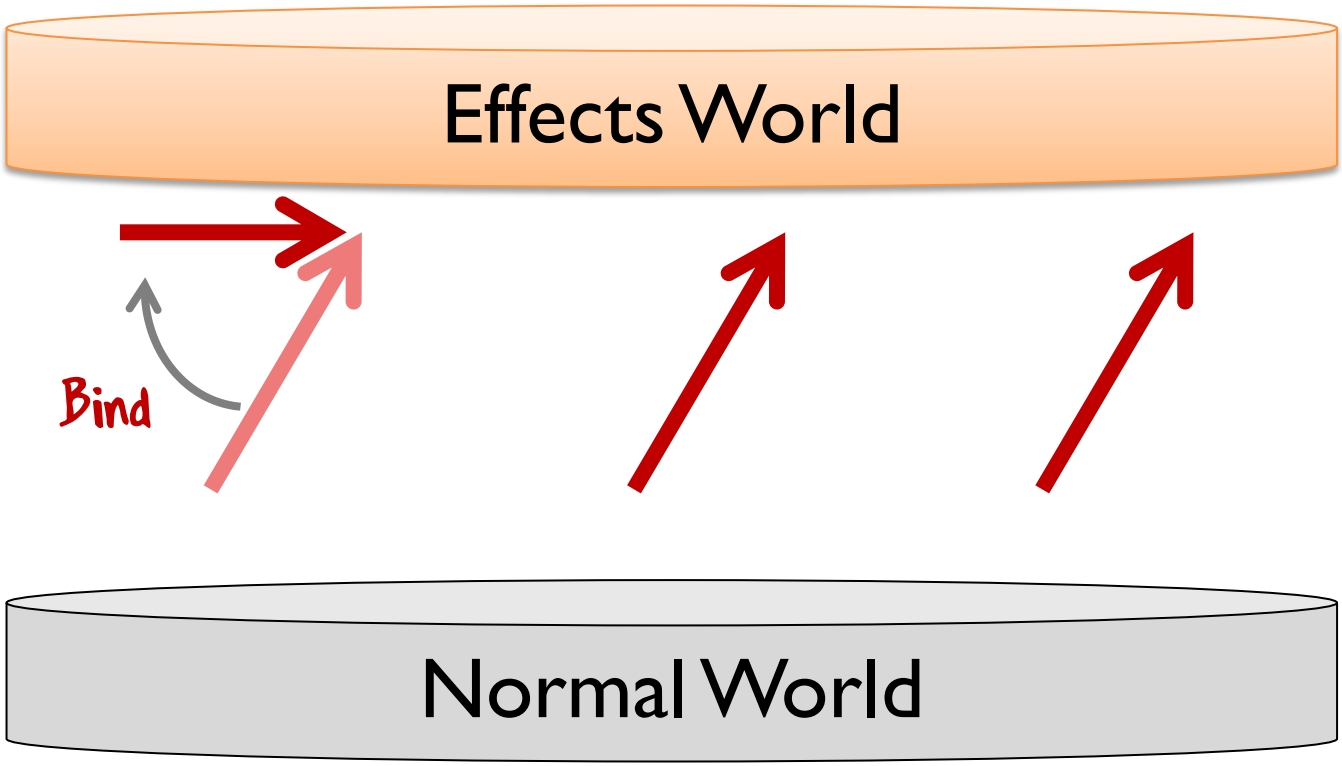


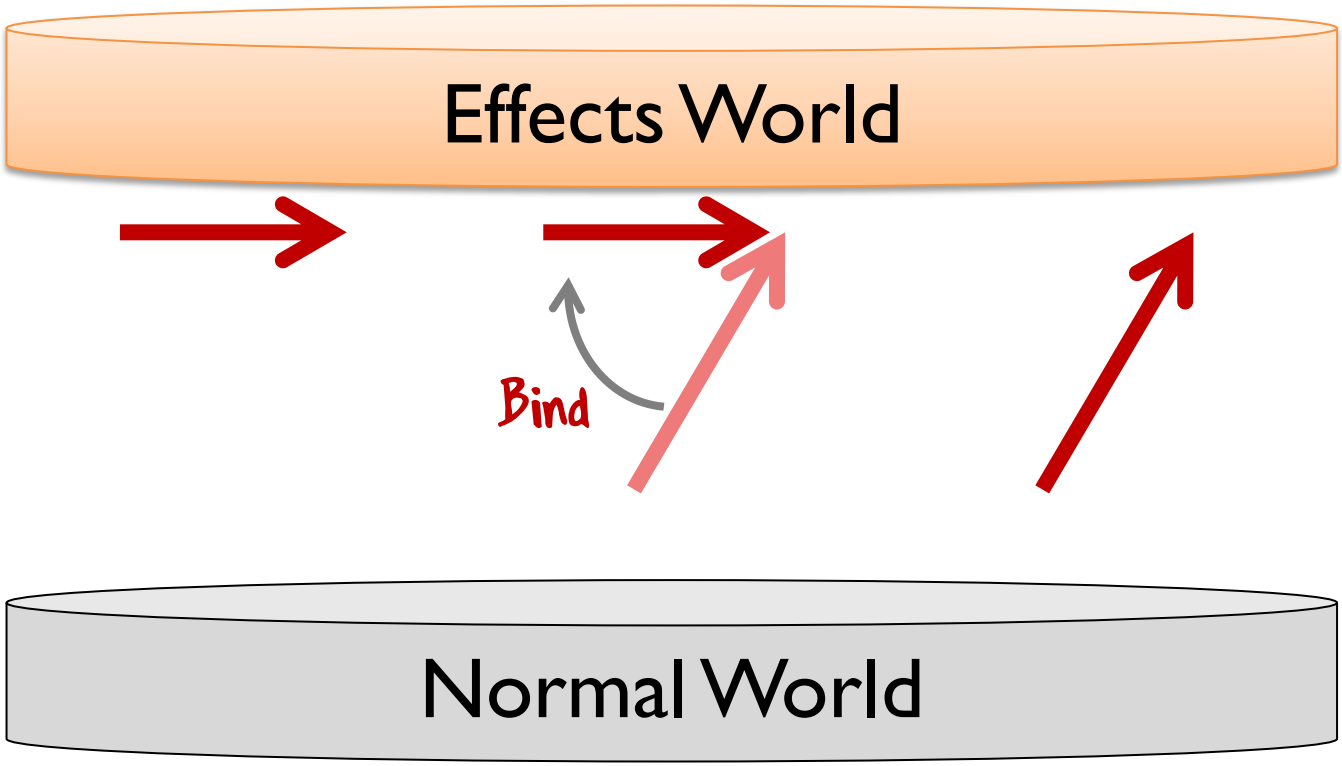
Before bind:
A diagonal function
(world crossing)

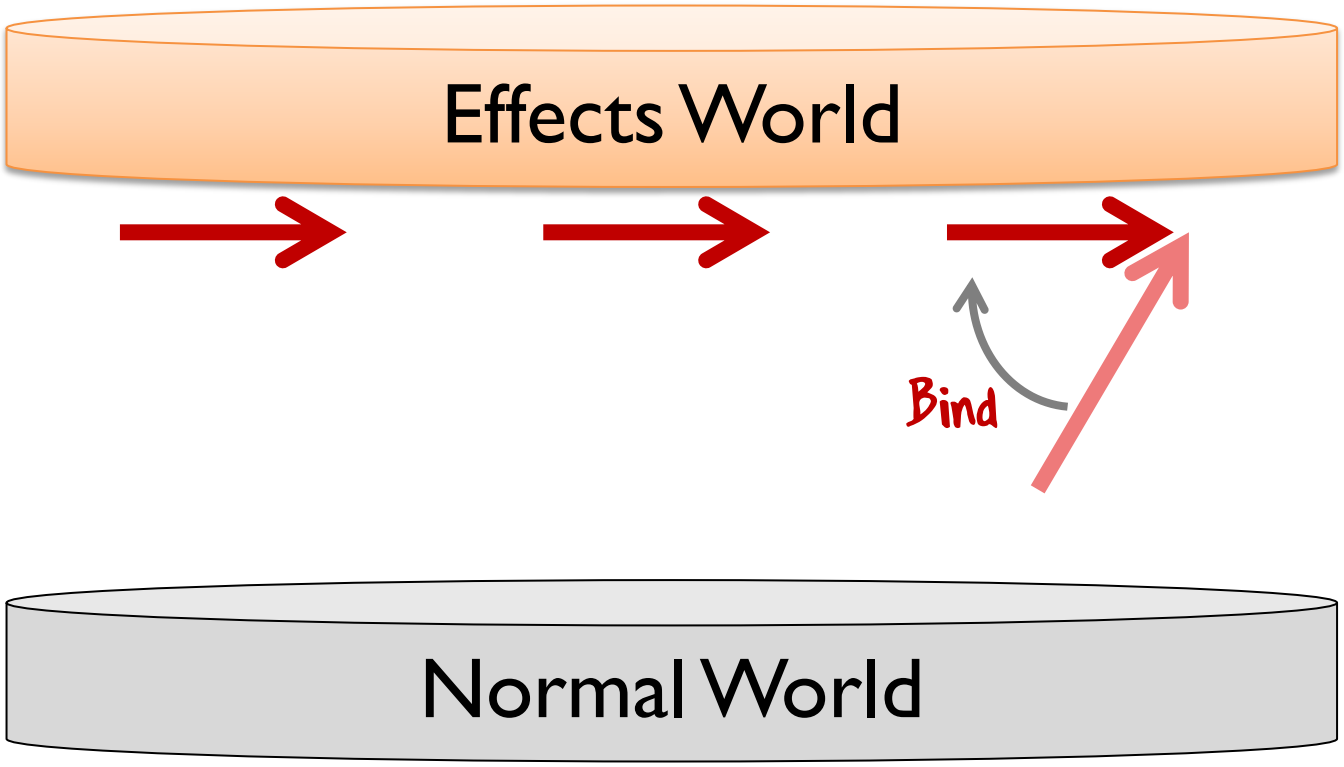


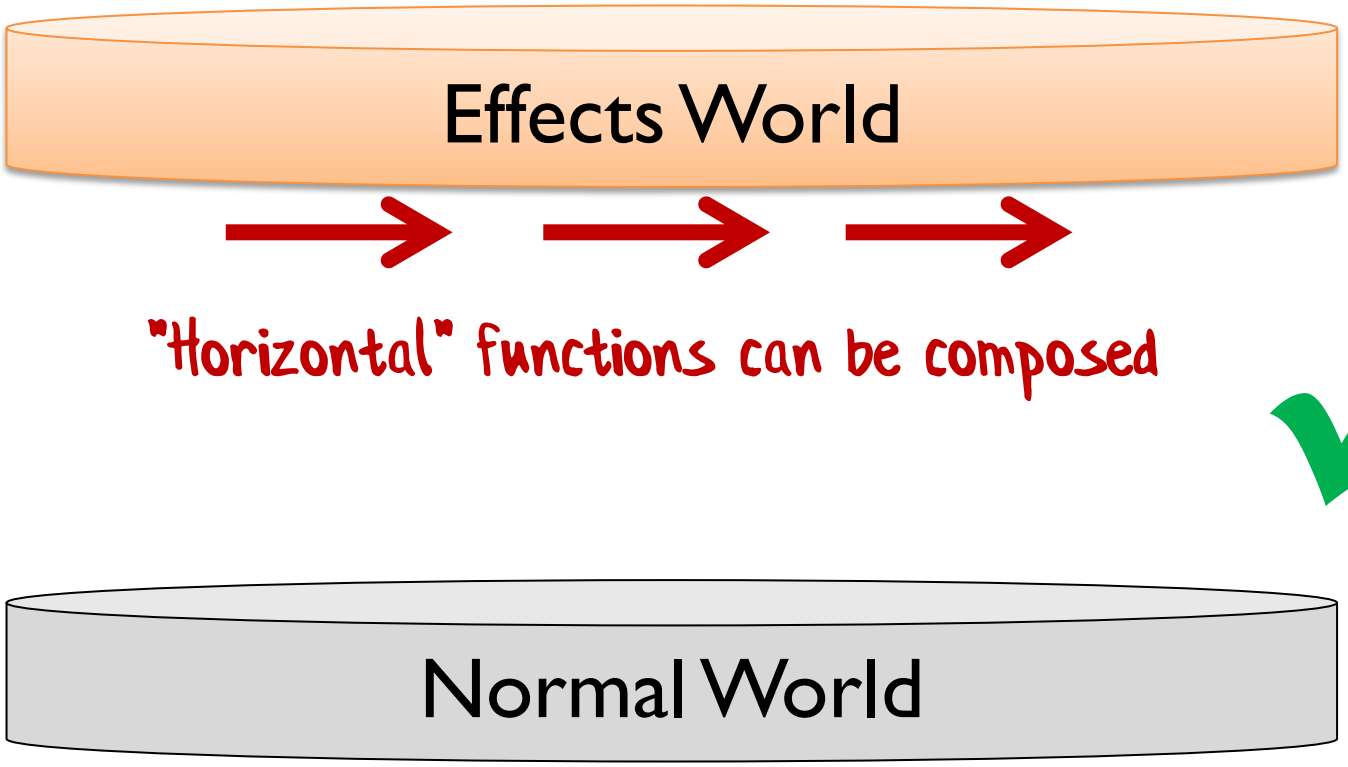
After bind:
A horizontal function
(all in E-world)


Effects World
"Diagonal" functions can't be composed
Normal World


Effects World
Bind
Normal World


Effects World
Bind
Normal World


Effects World
Bind
Normal World


Effects World
"Horizontal" functions can be composed
Normal World

FP terminology

A **monad** is

i. An effect type
   - e.g. `Option<>, List<>, Async<>`

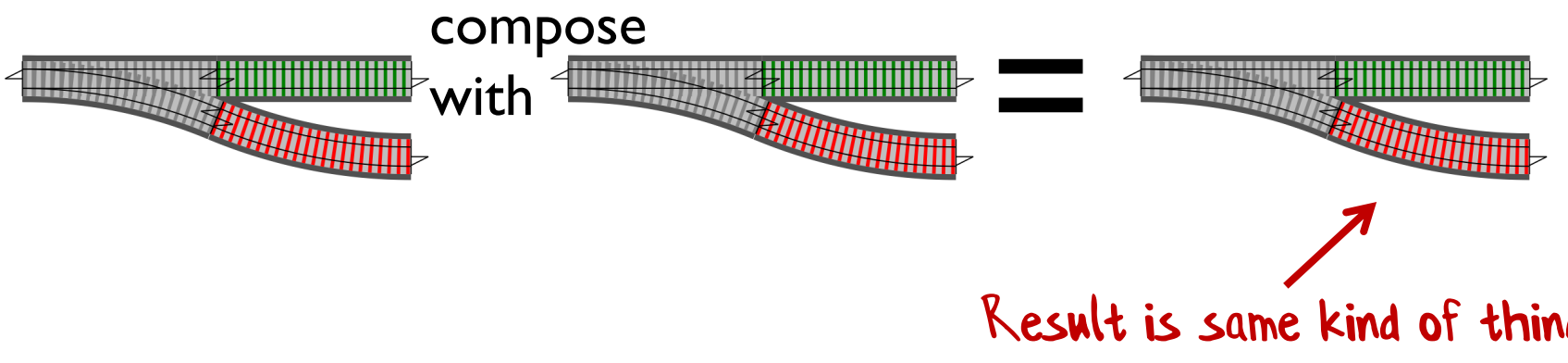
ii. Plus a `return` function
   - a.k.a. `pure unit`

iii. Plus a `bind` function that converts a "diagonal" (world-crossing) function into a "horizontal" (E-world-only) function
   - a.k.a. `>>= flatMap SelectMany`

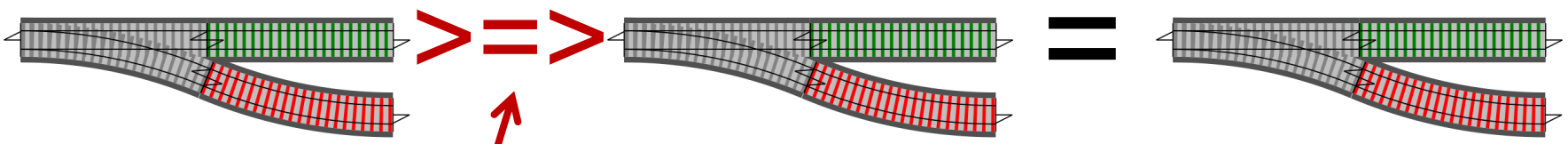
iv. And bind/return must have sensible implementations
   - the Monad laws

TLDR: If you want to chain effects-generating functions in series, use a **Monad**

Kleisli Composition



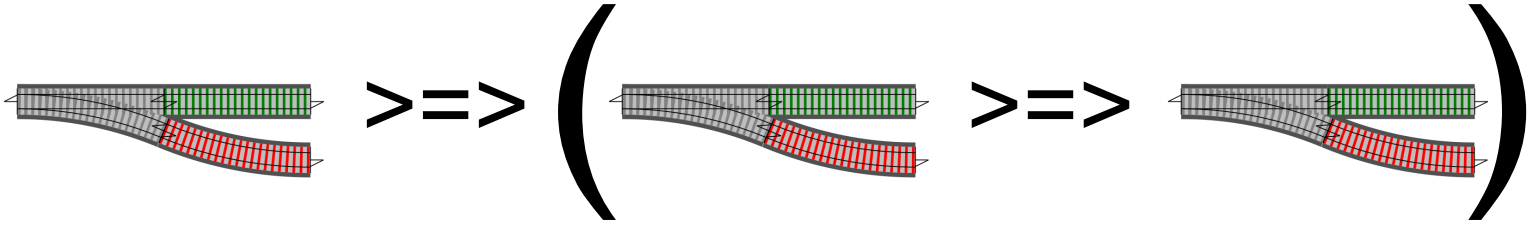
Kleisli Composition

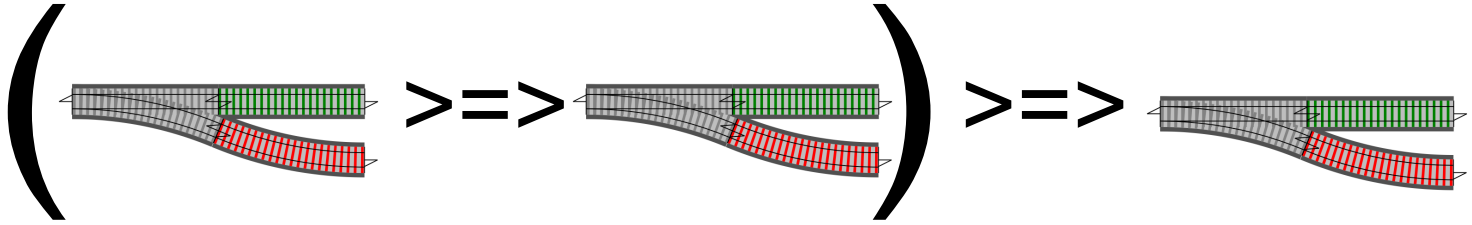


Kleisli composition symbol

Result is another monadic function so you can repeat.

Sound familiar?





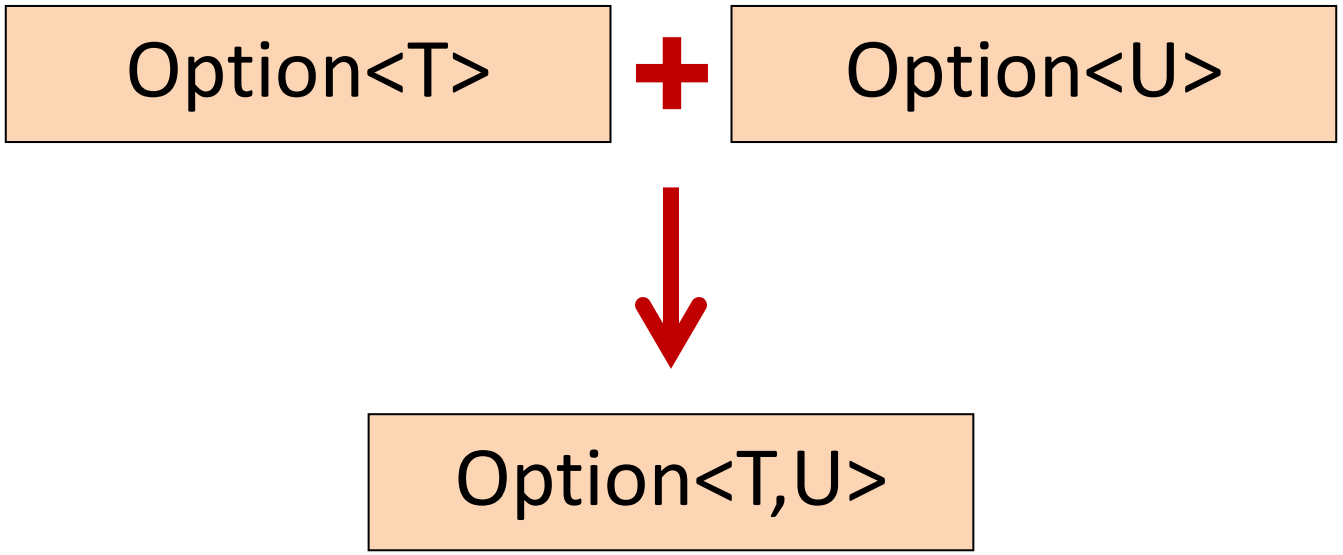
Order not important (Associative)

Monoid!

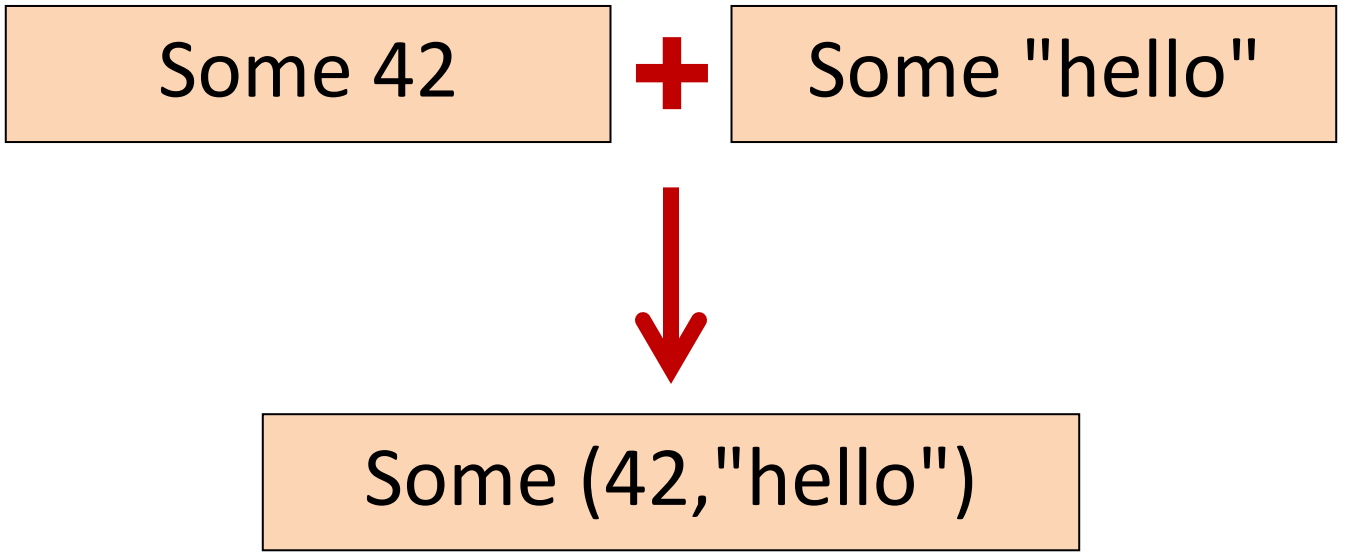
Tool #5

Combining effects in parallel with applicatives

How to combine effects?

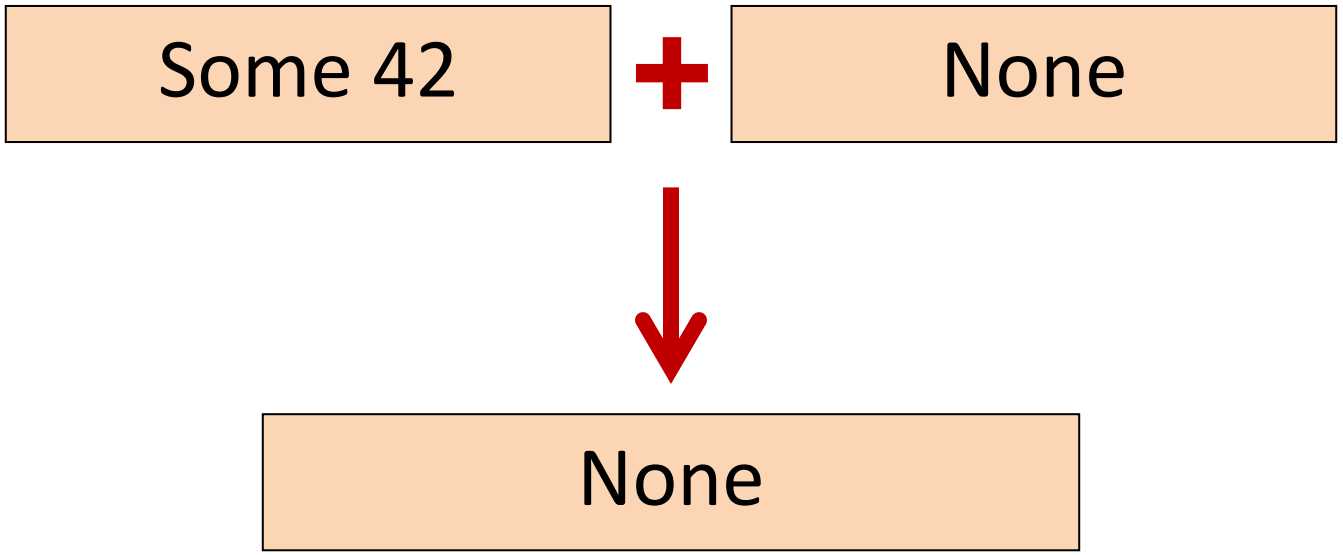


Combining options

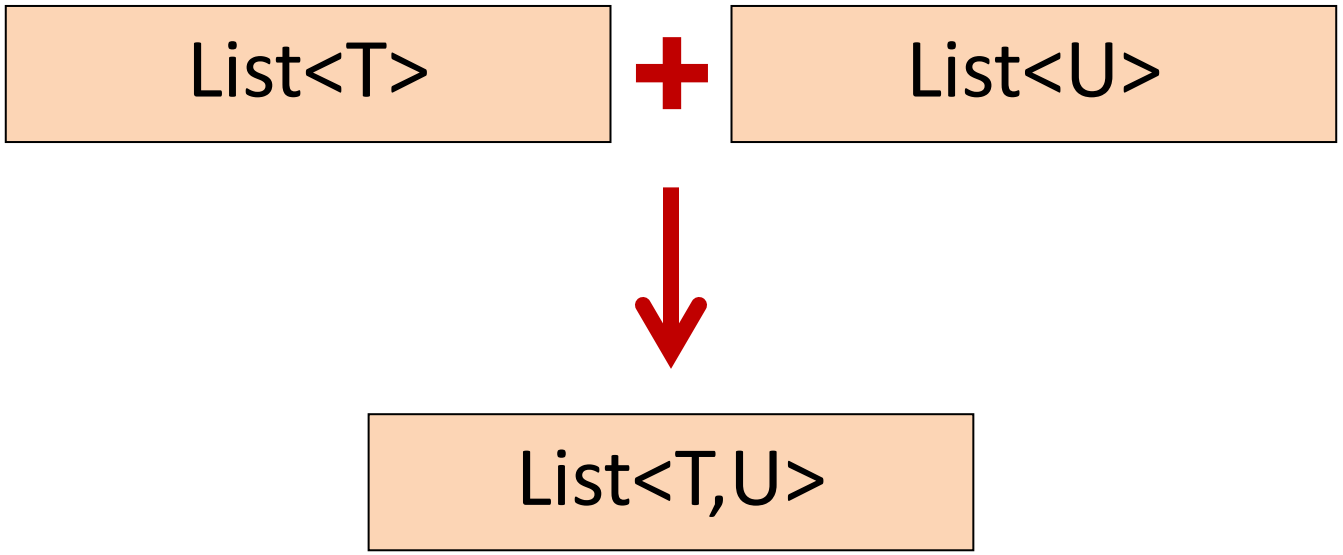


This is what you expect!

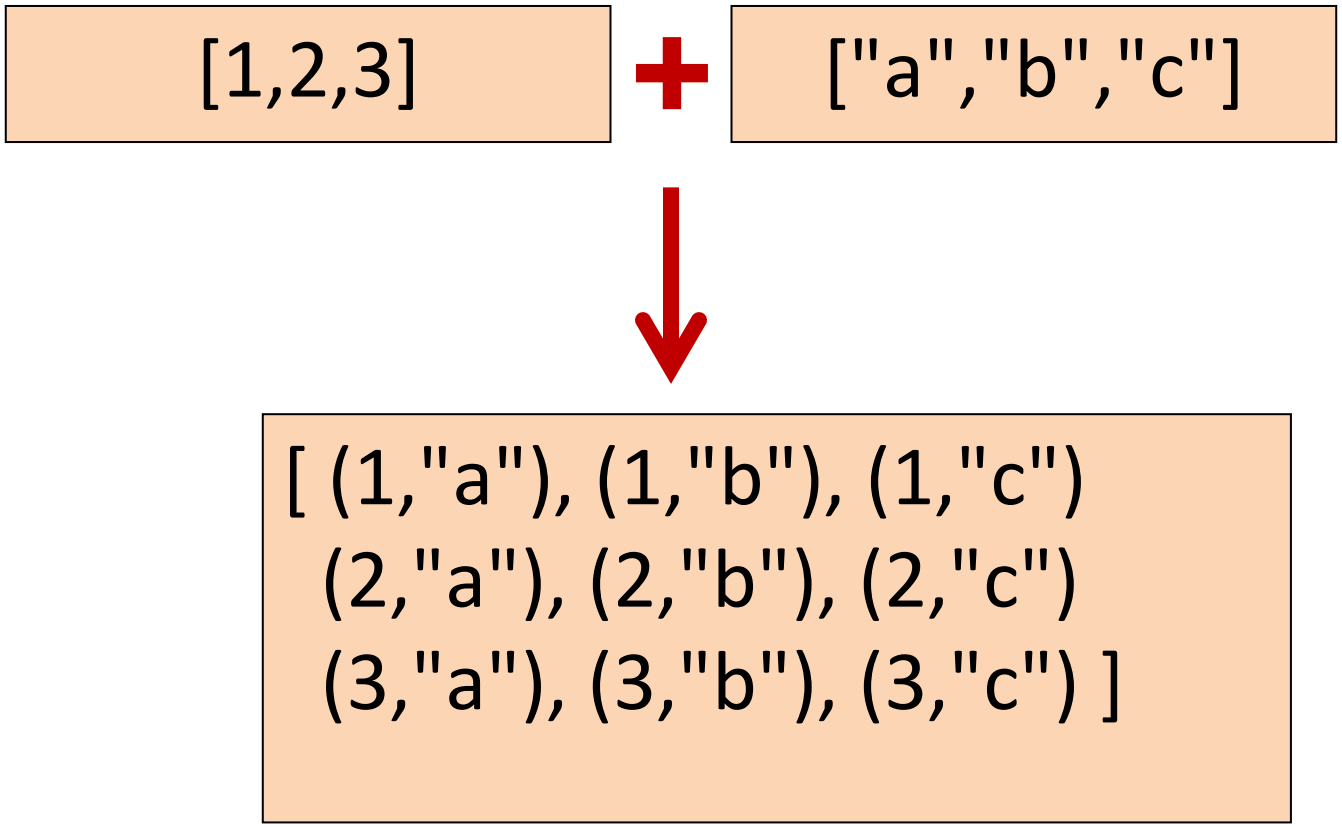
Combining options



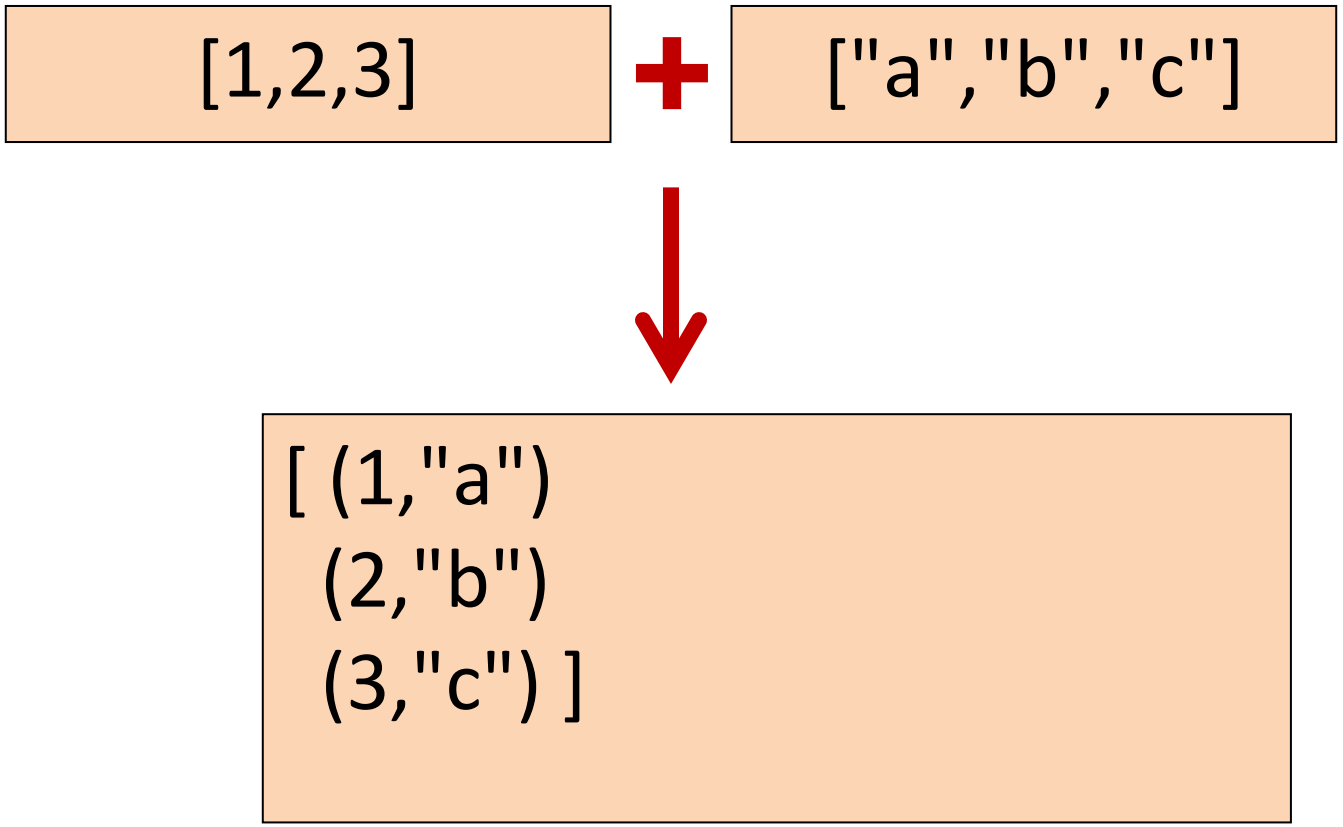
How to combine Lists?



Combining lists (cross product)



Combining lists (zip)



The general term for this is "applicative functor"

Option, List, Async are all applicatives

FP terminology

So why is this useful?

A **applicative (functor)** is

i. An effect type
   - e.g. `Option<>, List<>, Async<>`

ii. Plus a `return` function
   - a.k.a. `pure unit`

iii. Plus a function that combines two effects into one
   - a.k.a. `<*> apply pair`

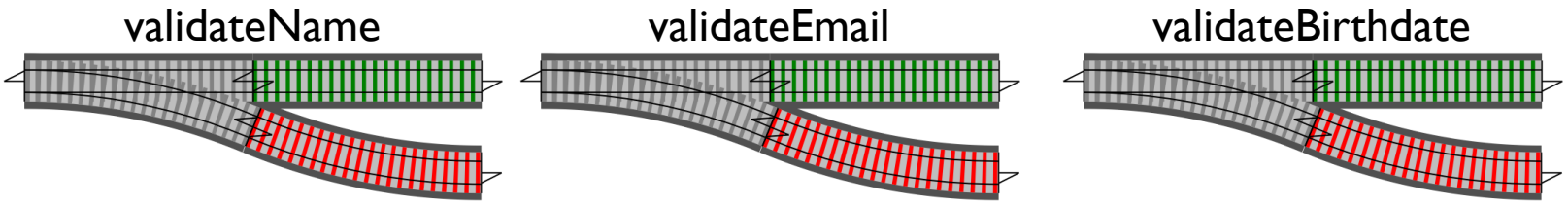
iv. And apply/return must have sensible implementations
   - the Applicative Functor laws

Problem:
How to validate multiple fields in parallel?

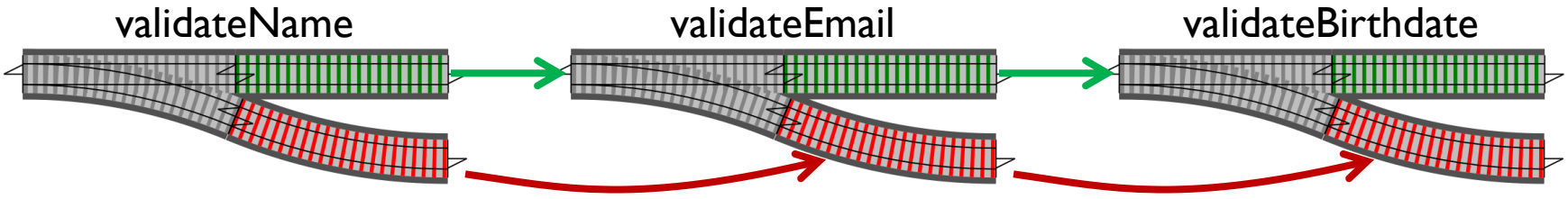
```
type Customer = {
  Name : String50
  Email : EmailAddress
  Birthdate : Date
}
```

Each field must be validated

So we create some validation functions:

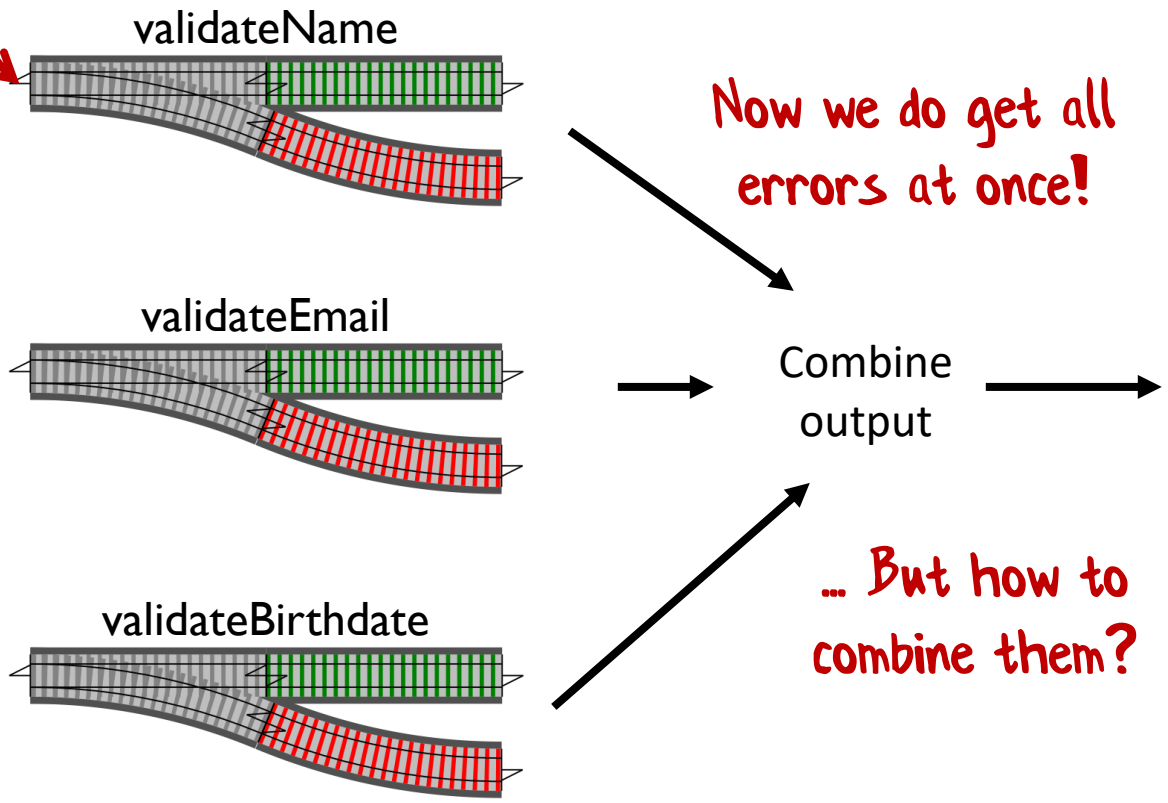


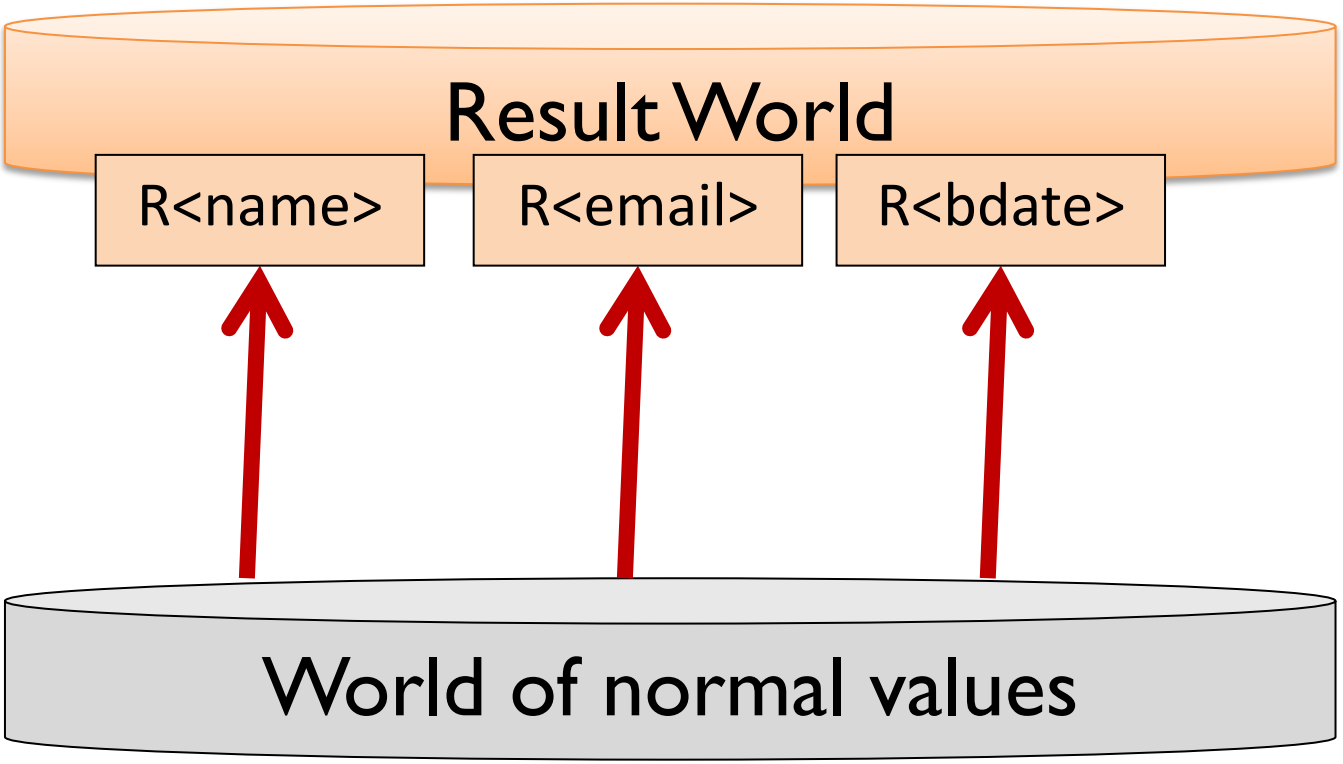
Problem: Validation done in series.
So only one error at a time is returned



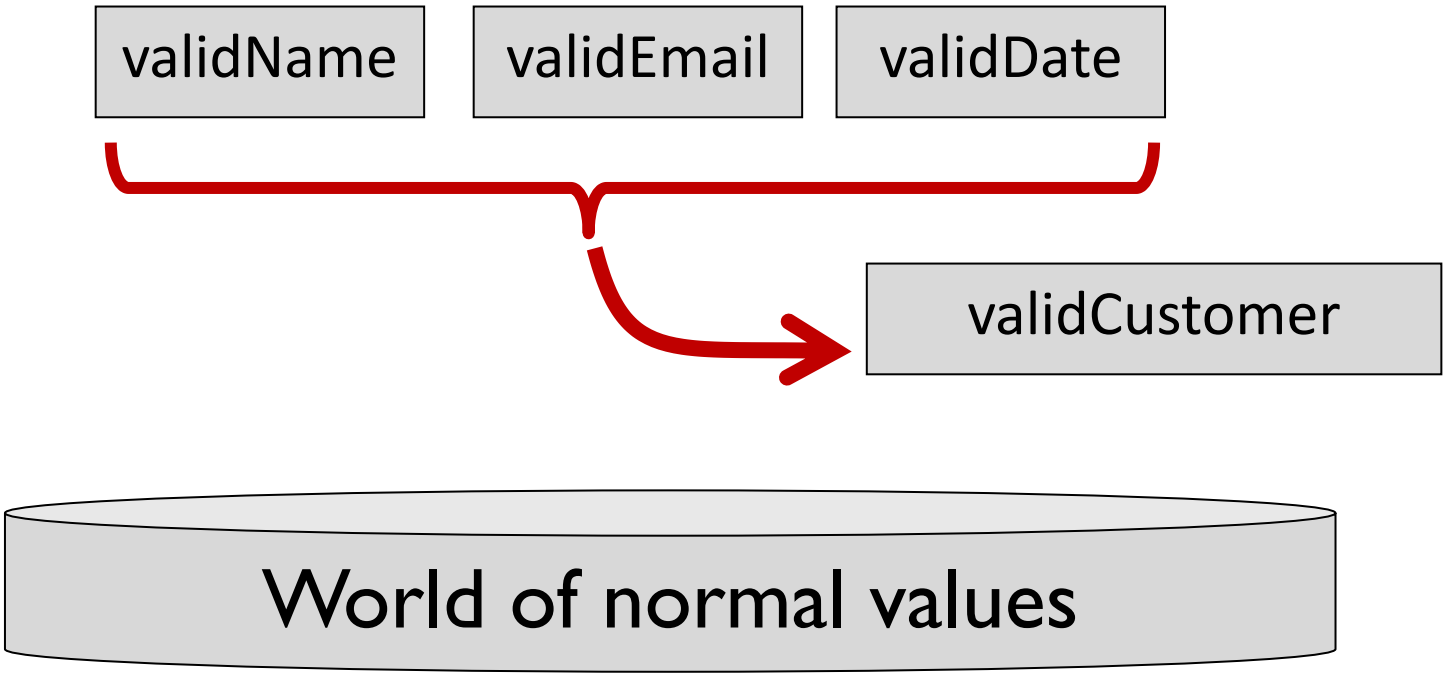
It would be nice to return all validation errors at once.

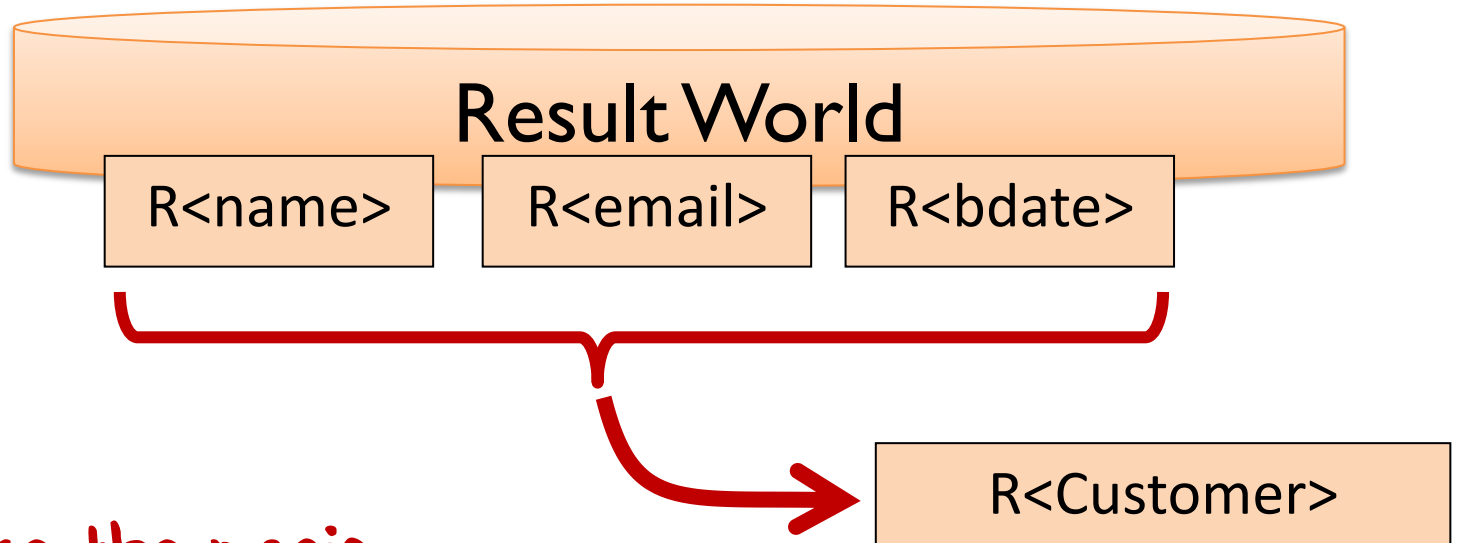
```
type CustomerDto = {
  name : string
  email : string
  birthdate : string
}
```




Result World
R<name>
R<email>
R<bdate>
World of normal values

We know how to combine the normal values
(use a constructor)





Use the magic of Applicatives!

The output is also in Result world

What the code looks like

```
let createValidCustomer (dto:CustomerDto) =
  // get the validated values
  let nameOrError = validateName dto.name
  let emailOrError = validateEmail dto.email
  let birthdateOrError =
    validateBirthdate dto.birthdate

  // call the constructor
  makeCustomer
    <!> nameOrError
    <*> emailOrError
    <*> birthdateOrError

// final output is Result<Customer,ErrMsg list>
```

Here's where the magic happens!

Monoids used here

Let's review the tools

The Functional Toolbox

- "**combine**"
  - Combines two values to make another one of the same kind
- "**reduce**"
  - Reduces a list to a single value by using "combine" repeatedly

The Functional Toolbox

- "**map**"
  - Lifts **functions** into an effects world
- "**return**"
  - Lifts **values** into an effects world
- "**bind**"
  - Converts "diagonal" functions into "horizontal" ones so they can be composed.
- "**apply**"
  - Combines two effects in parallel
  - "**map2**", "**lift2**", "**lift3**" are specialized versions of "apply“

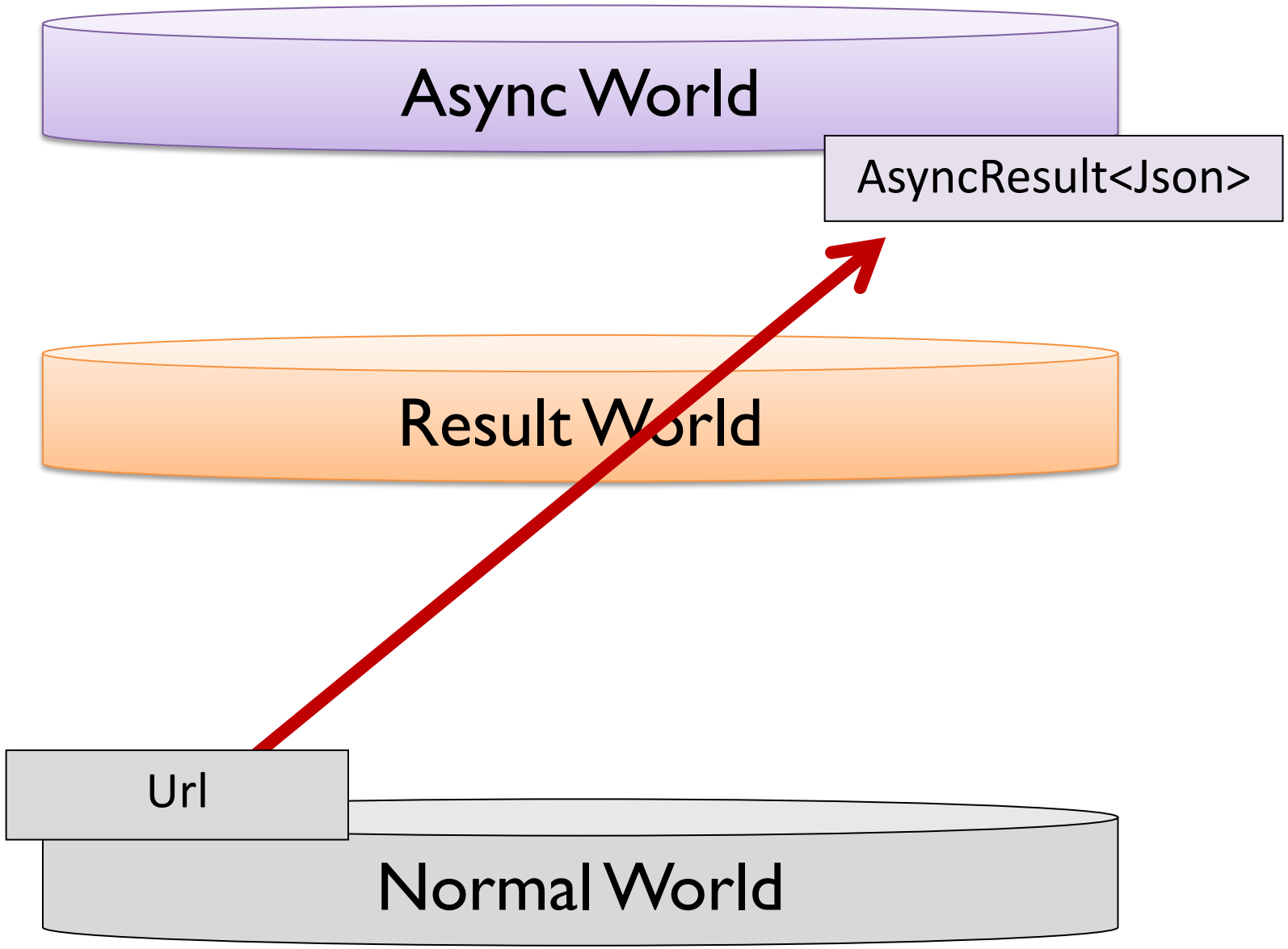
Example

Using all the tools together

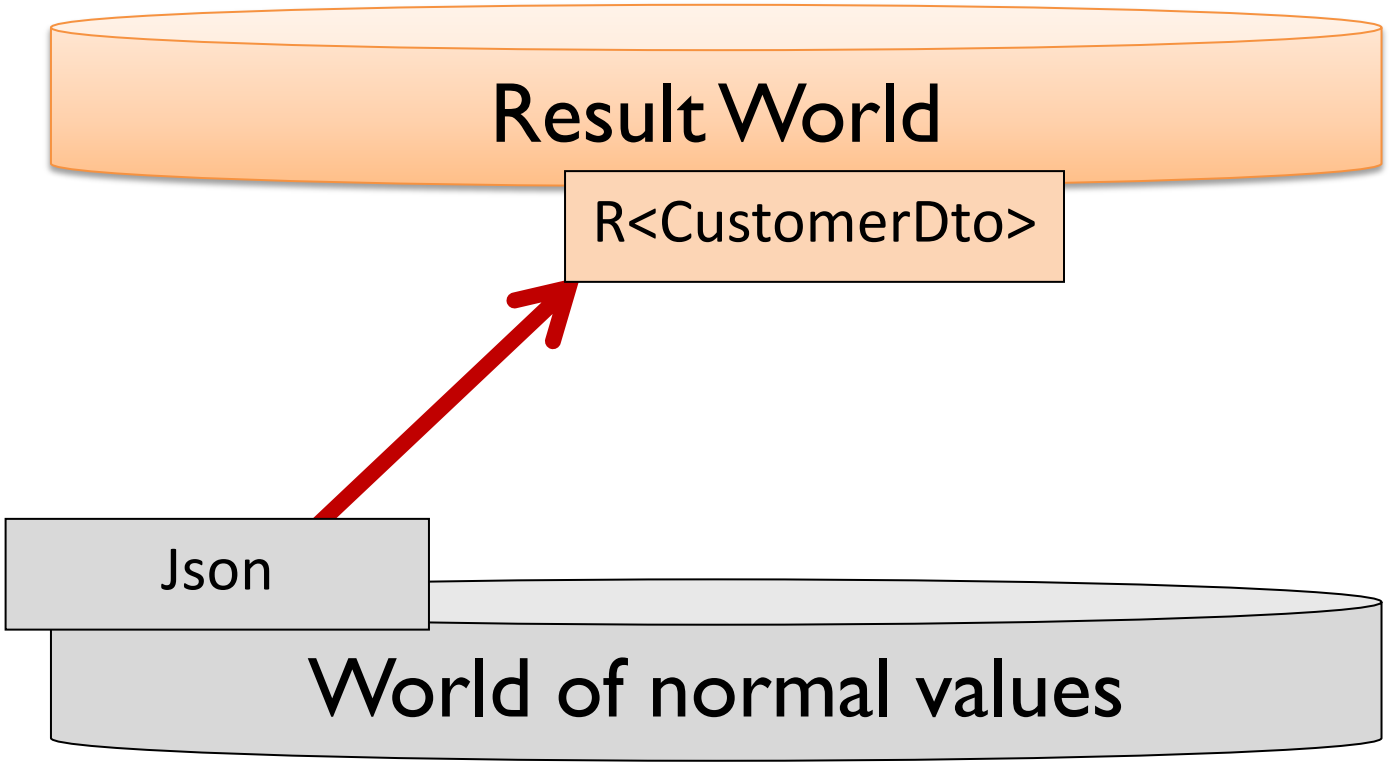
Example scenario

- Download a URL into a JSON file
- Decode the JSON into a Customer DTO
- Convert the DTO into a valid Customer
- Store the Customer in a database

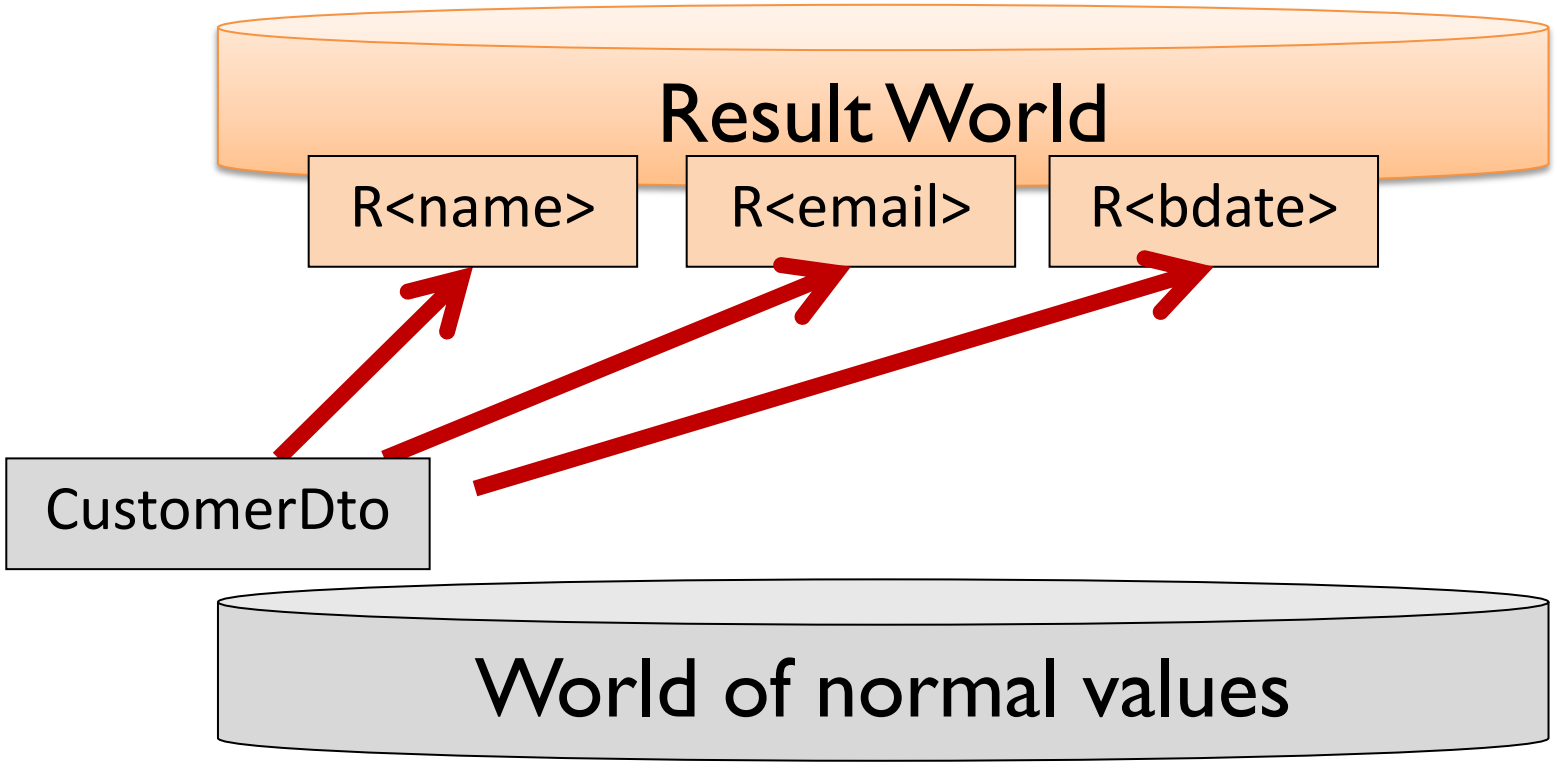
Download the json file



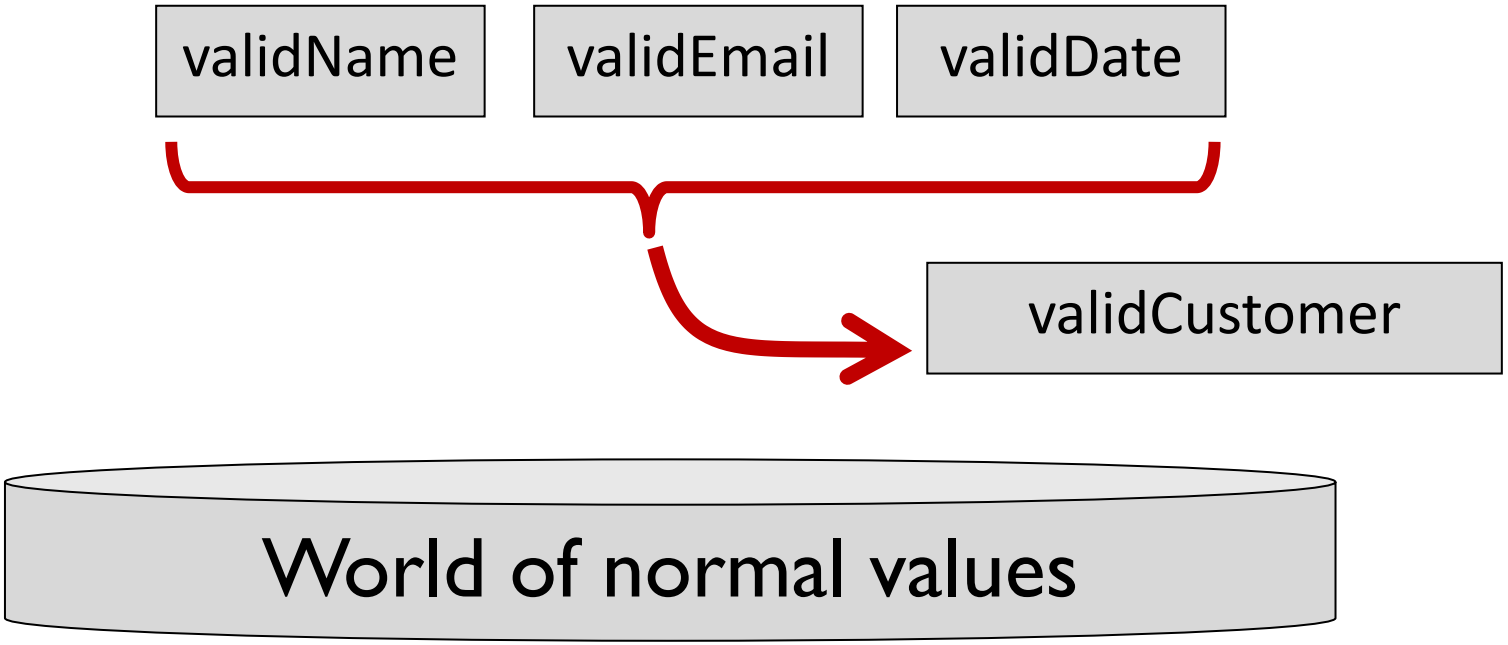
Decode the json



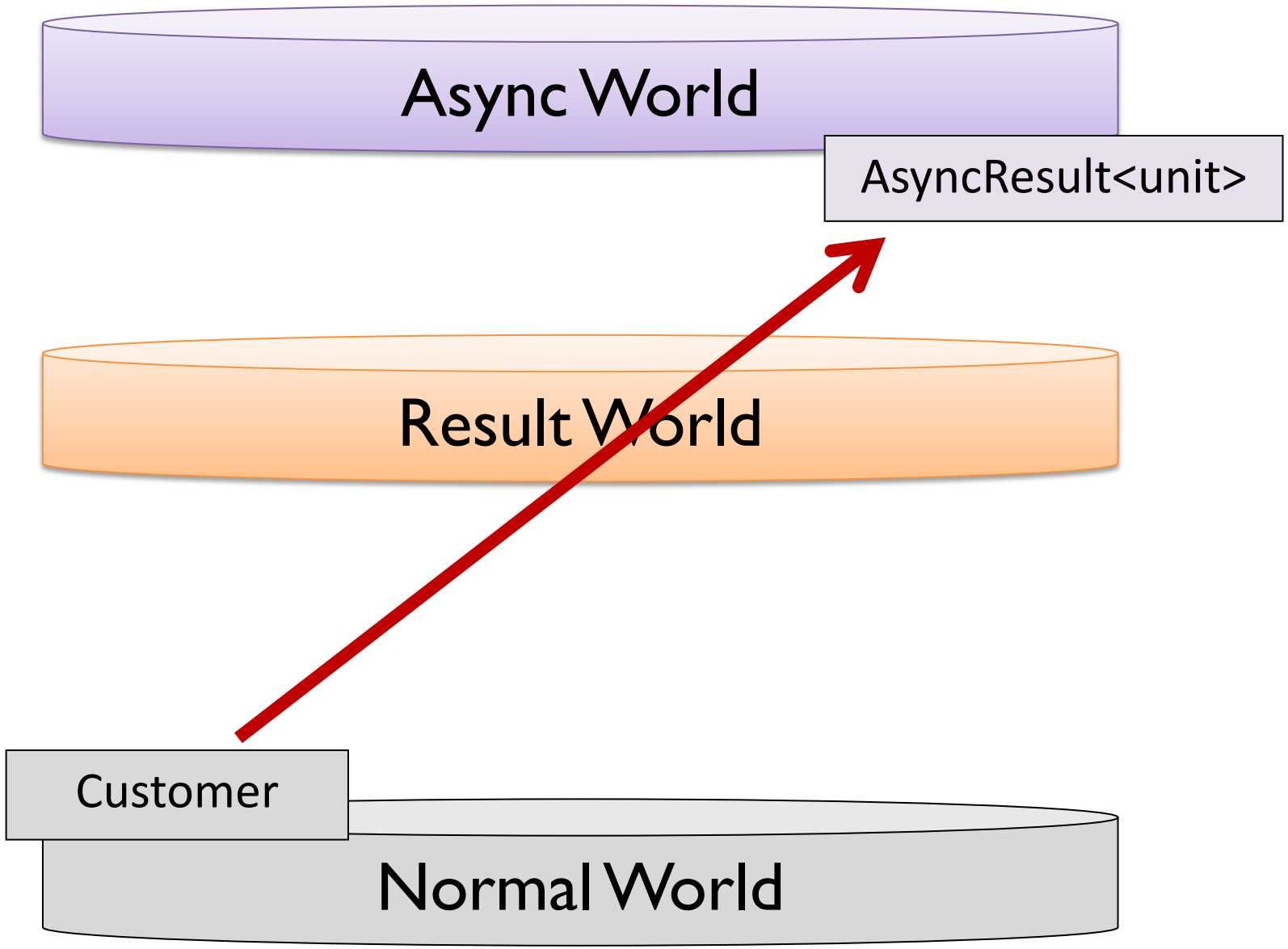
Validate fields



Construct the customer



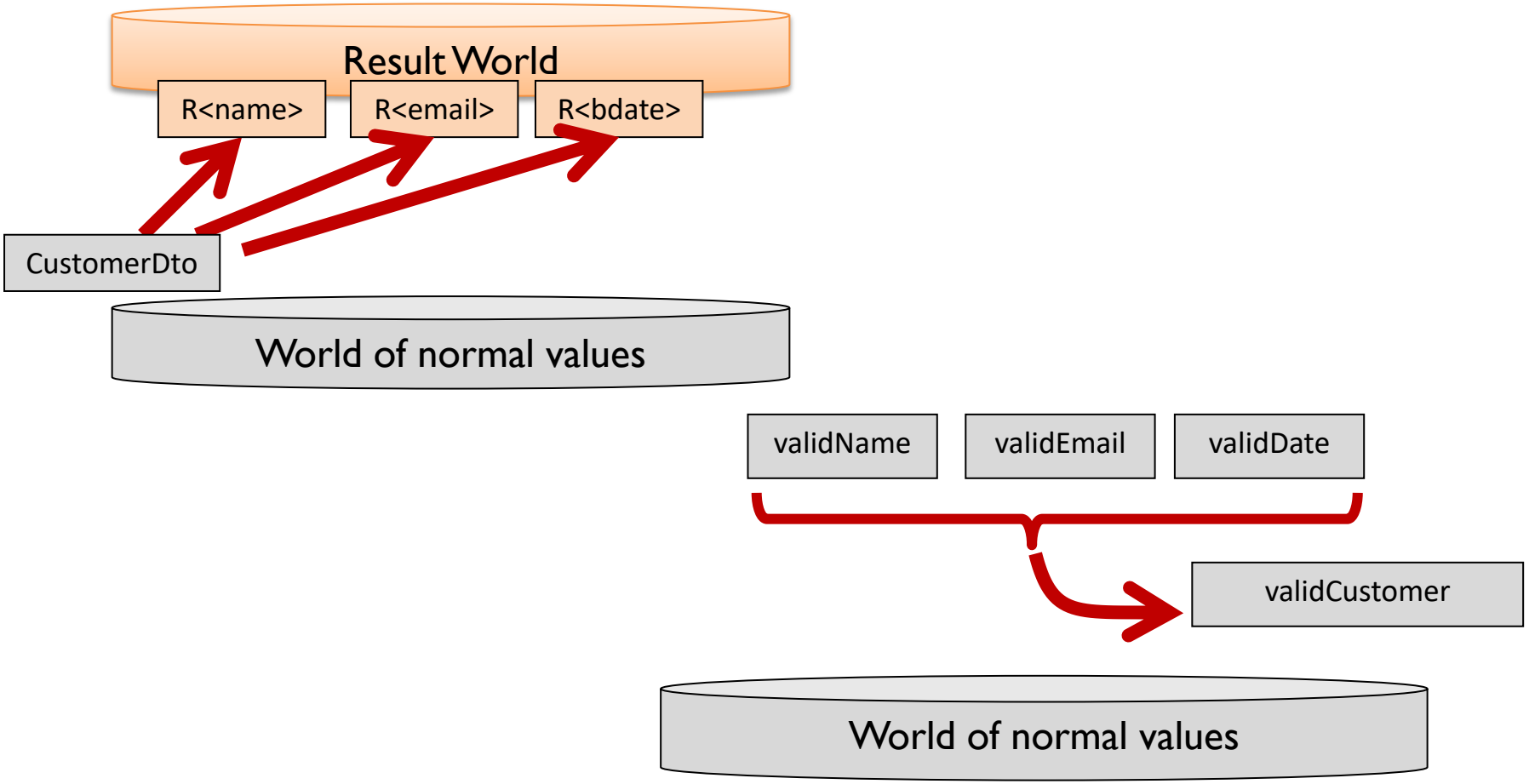
Store the customer



Now we DO have the tools to compose these functions together!

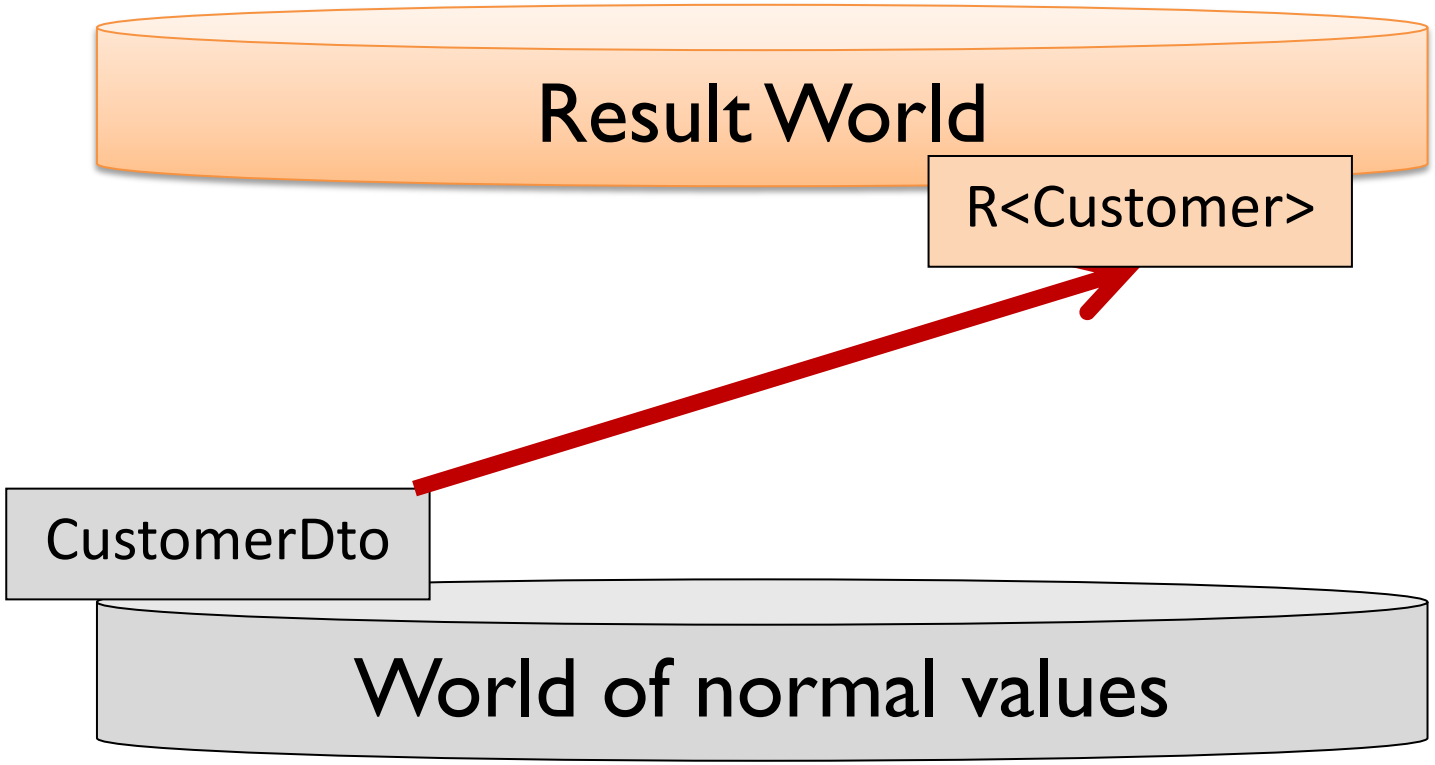
Validate fields AND create a customer

We already did this one using applicatives (and monoids)



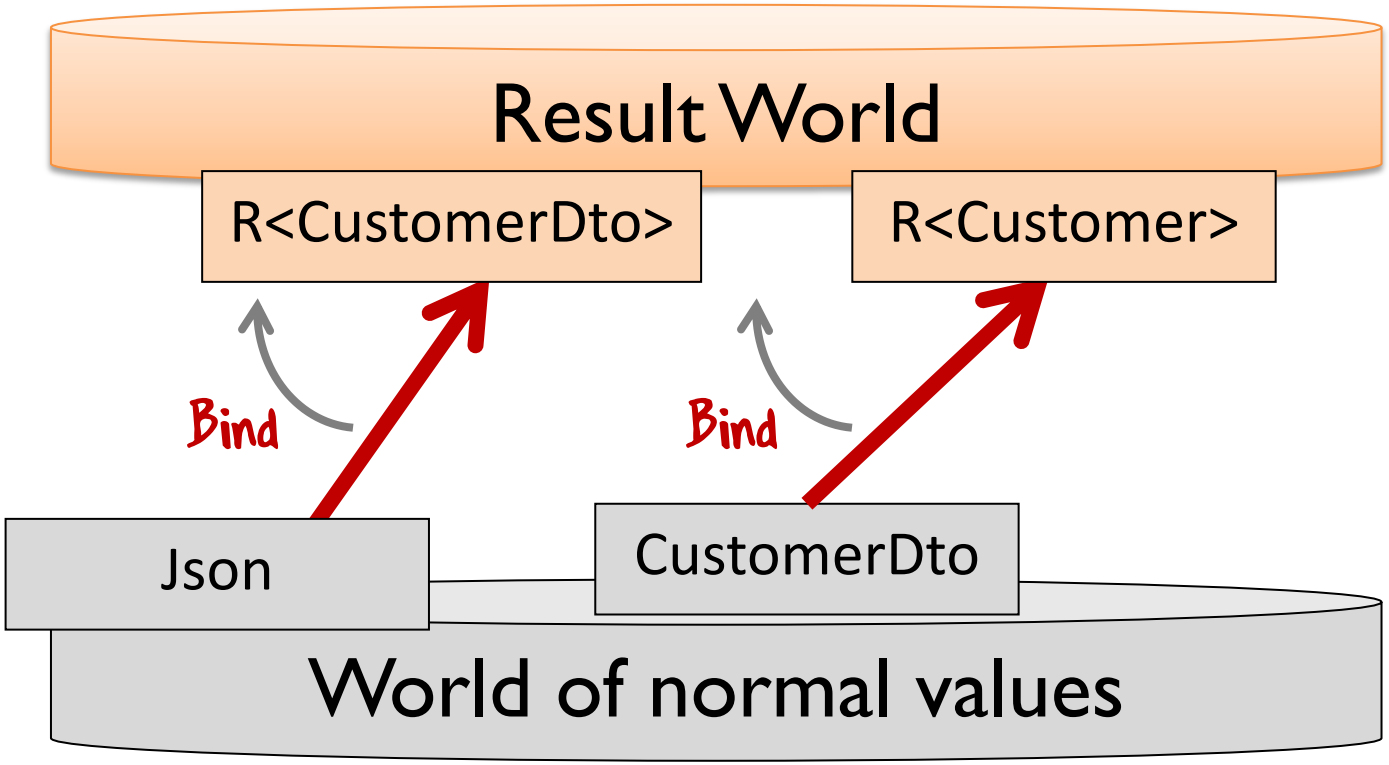
Validate fields AND create a customer

We already did this one using applicatives, and monoids

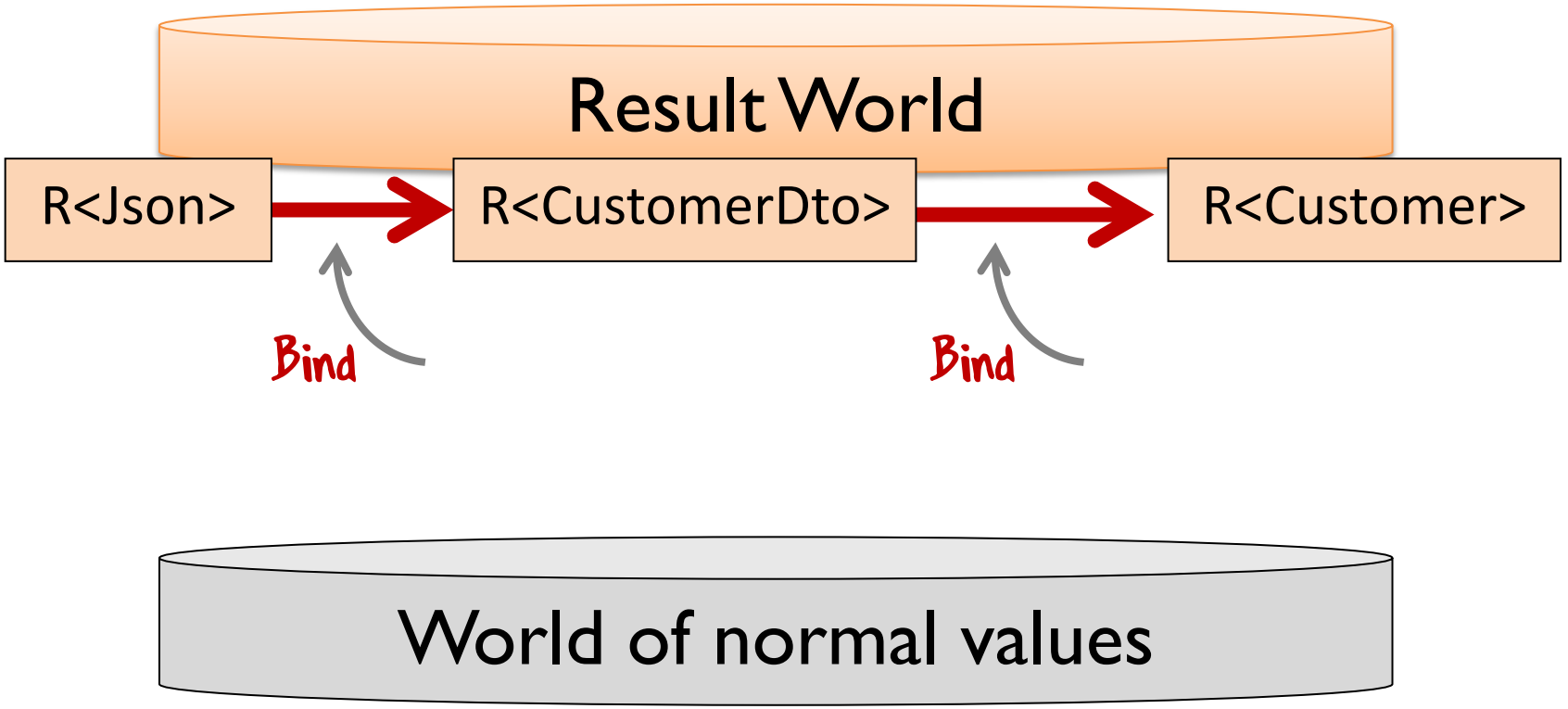


Parse json AND create a customer

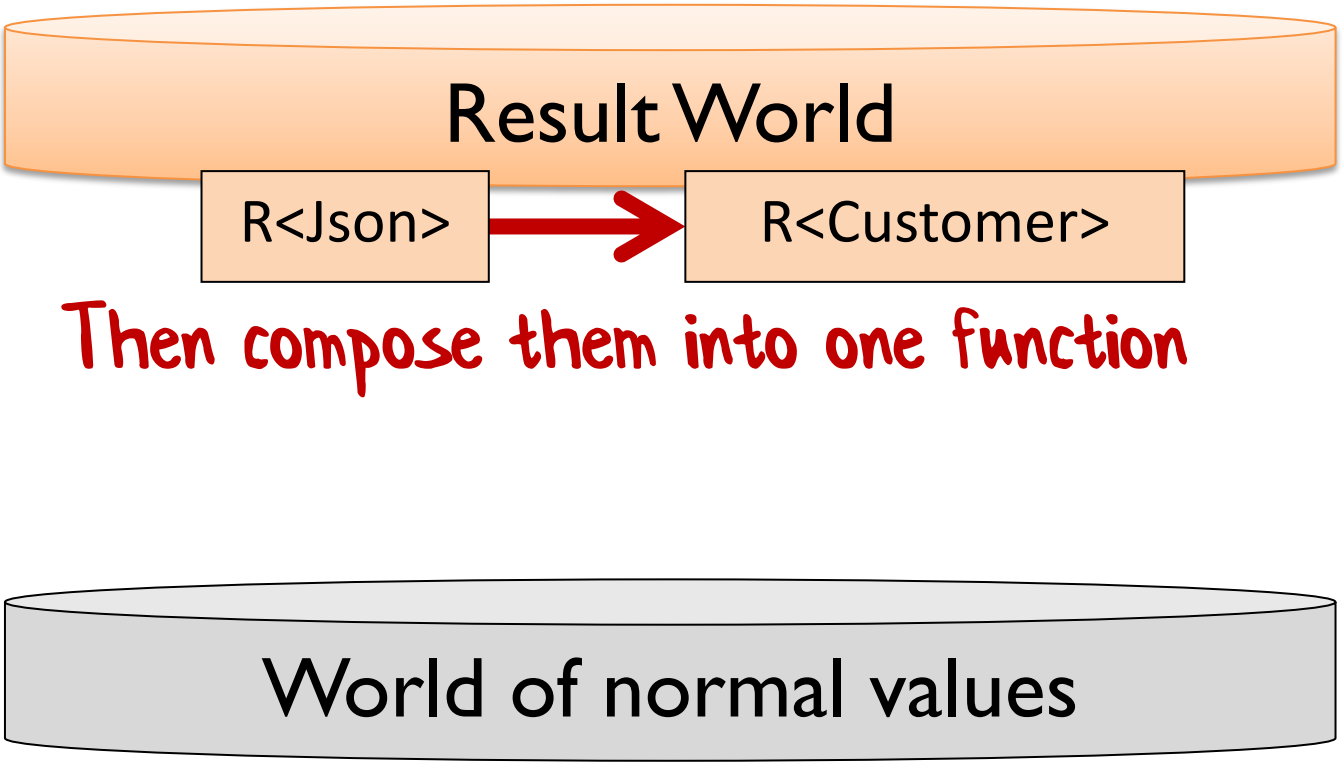
Use "bind" to turn the diagonal functions into horizontal ones



Parse json AND create a customer



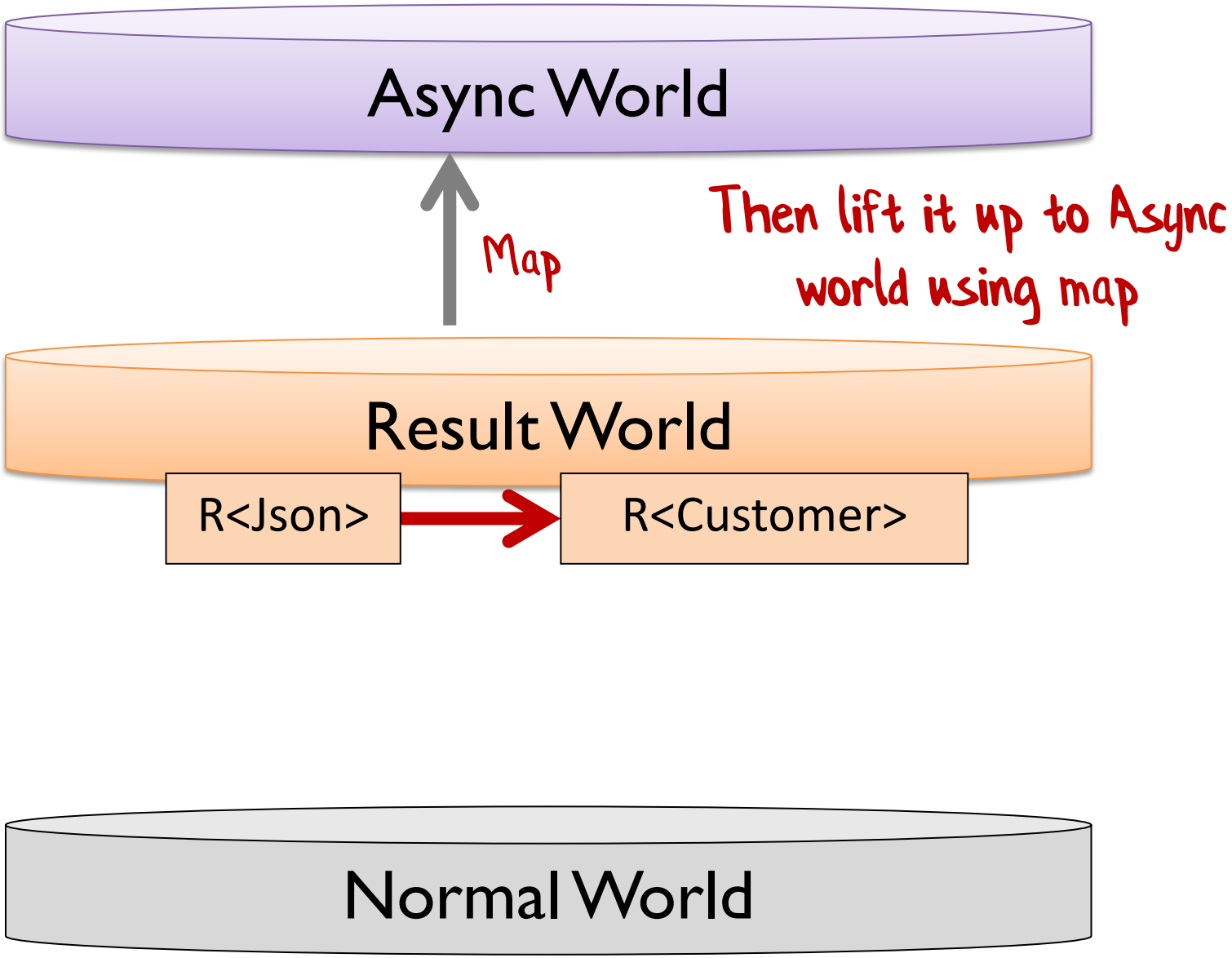
Parse json AND create a customer



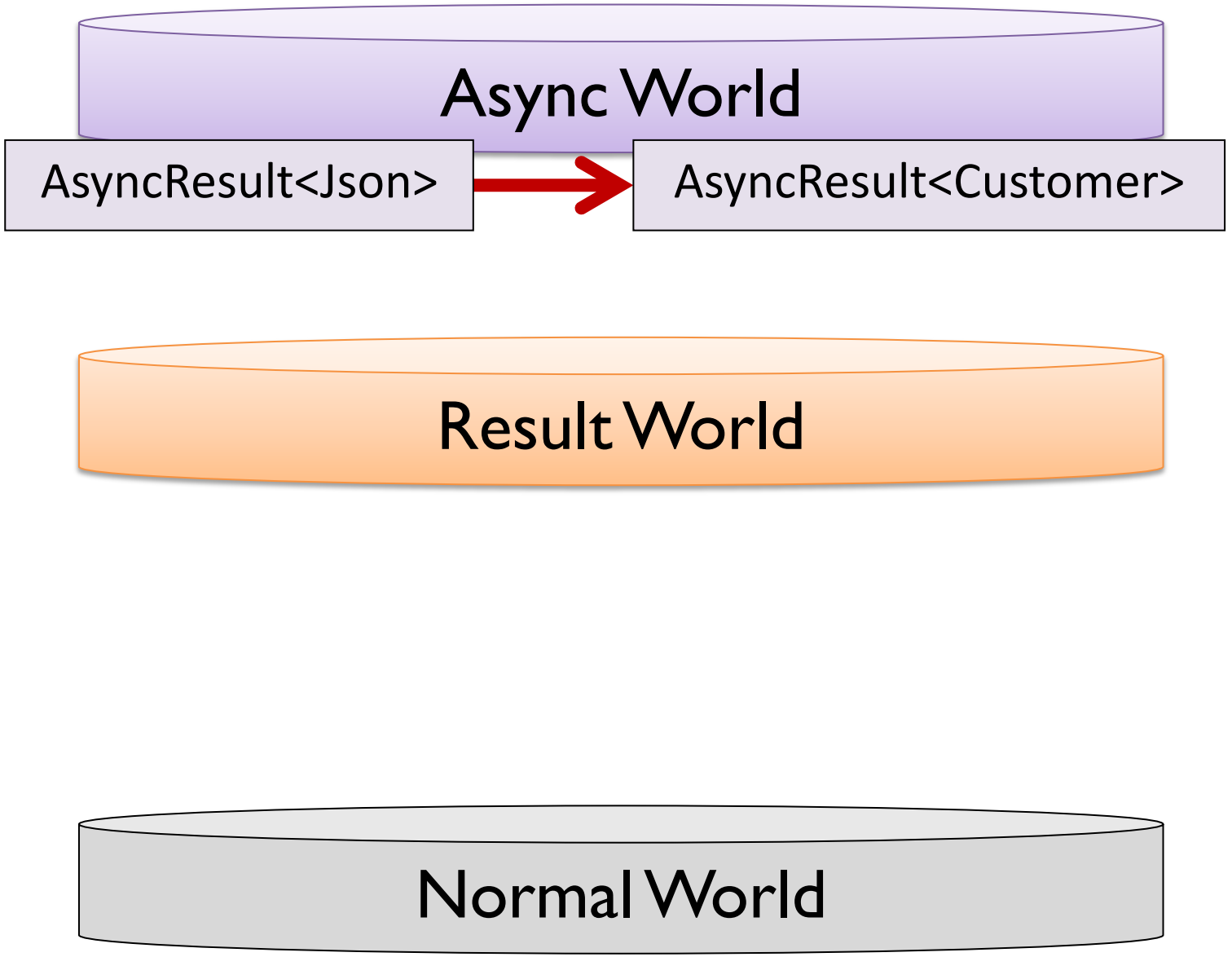
What the code looks like

```
let processCustomerDto jsonOrError =
  jsonOrError
  |> Result.bind decodeCustomerDto
  |> Result.bind createValidCustomer
```

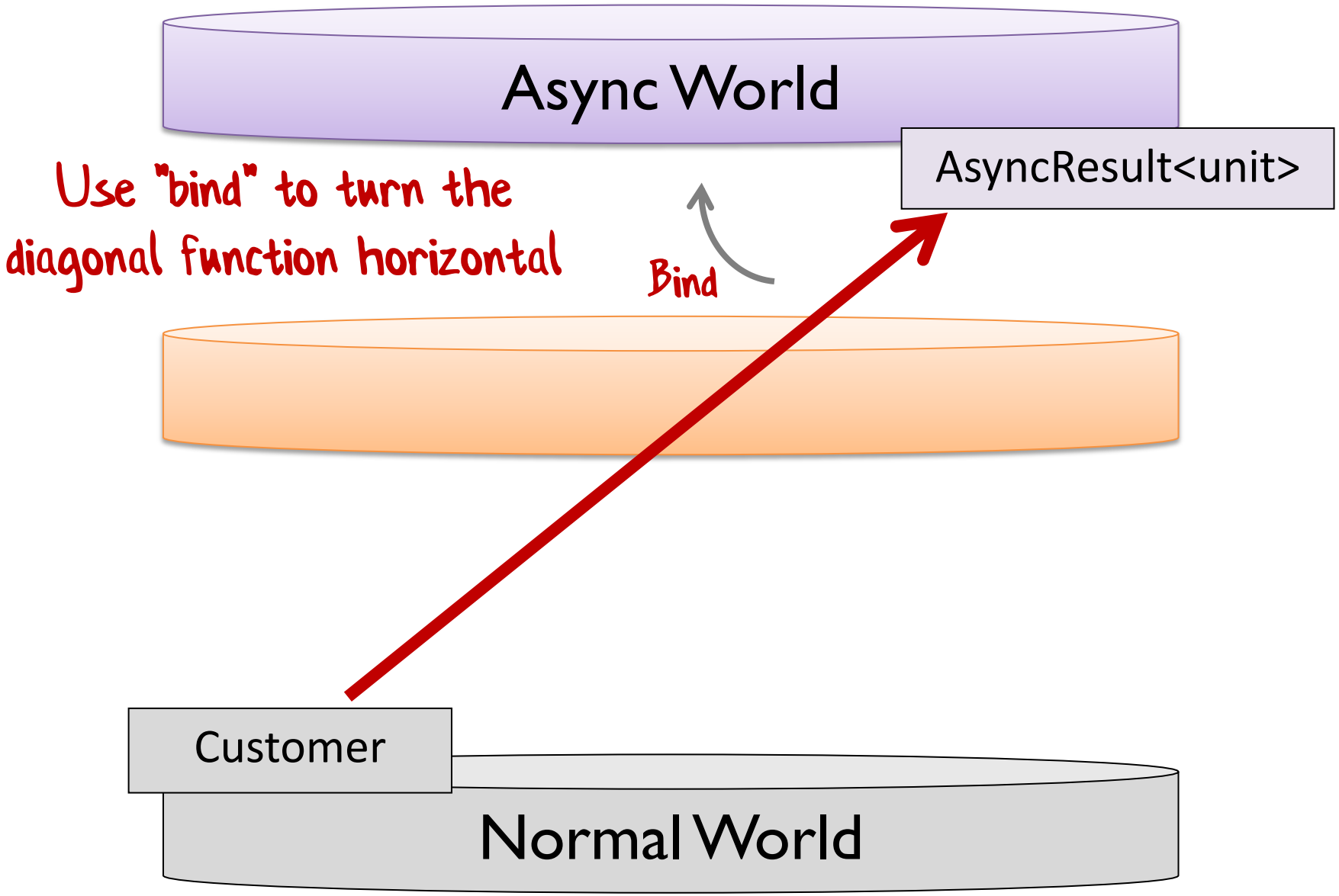
Parse json AND create a customer



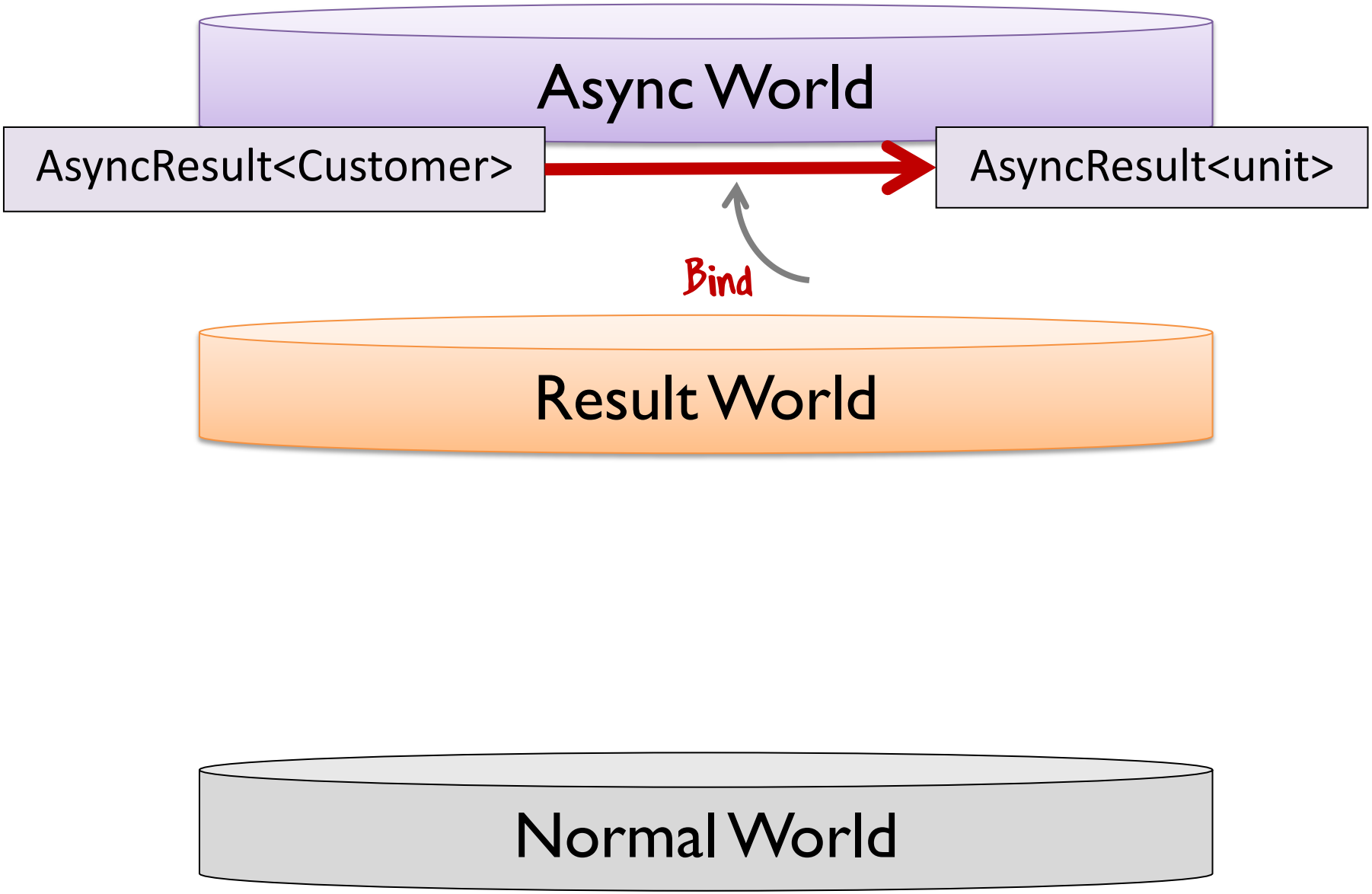
Parse json AND create a customer



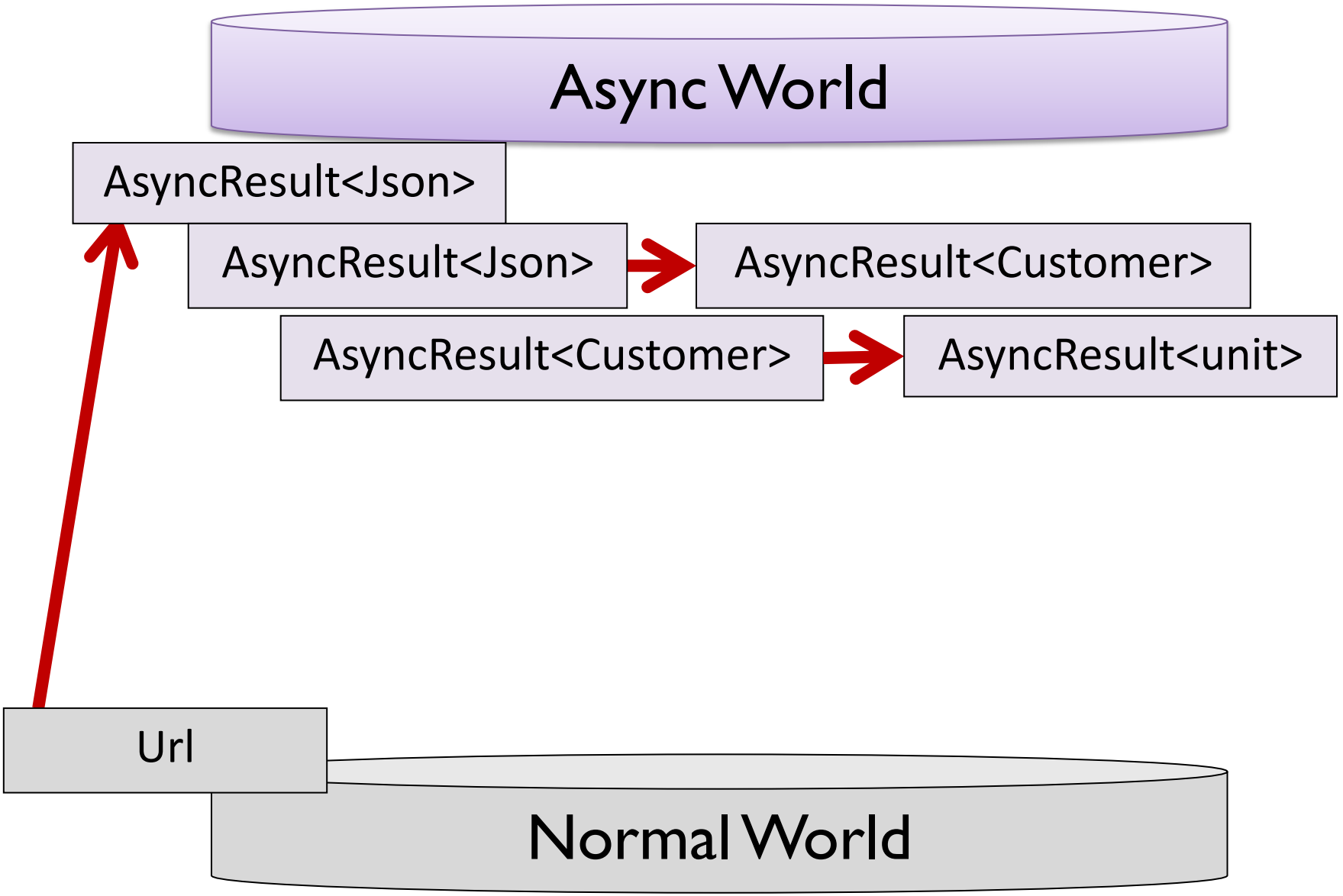
Store the customer



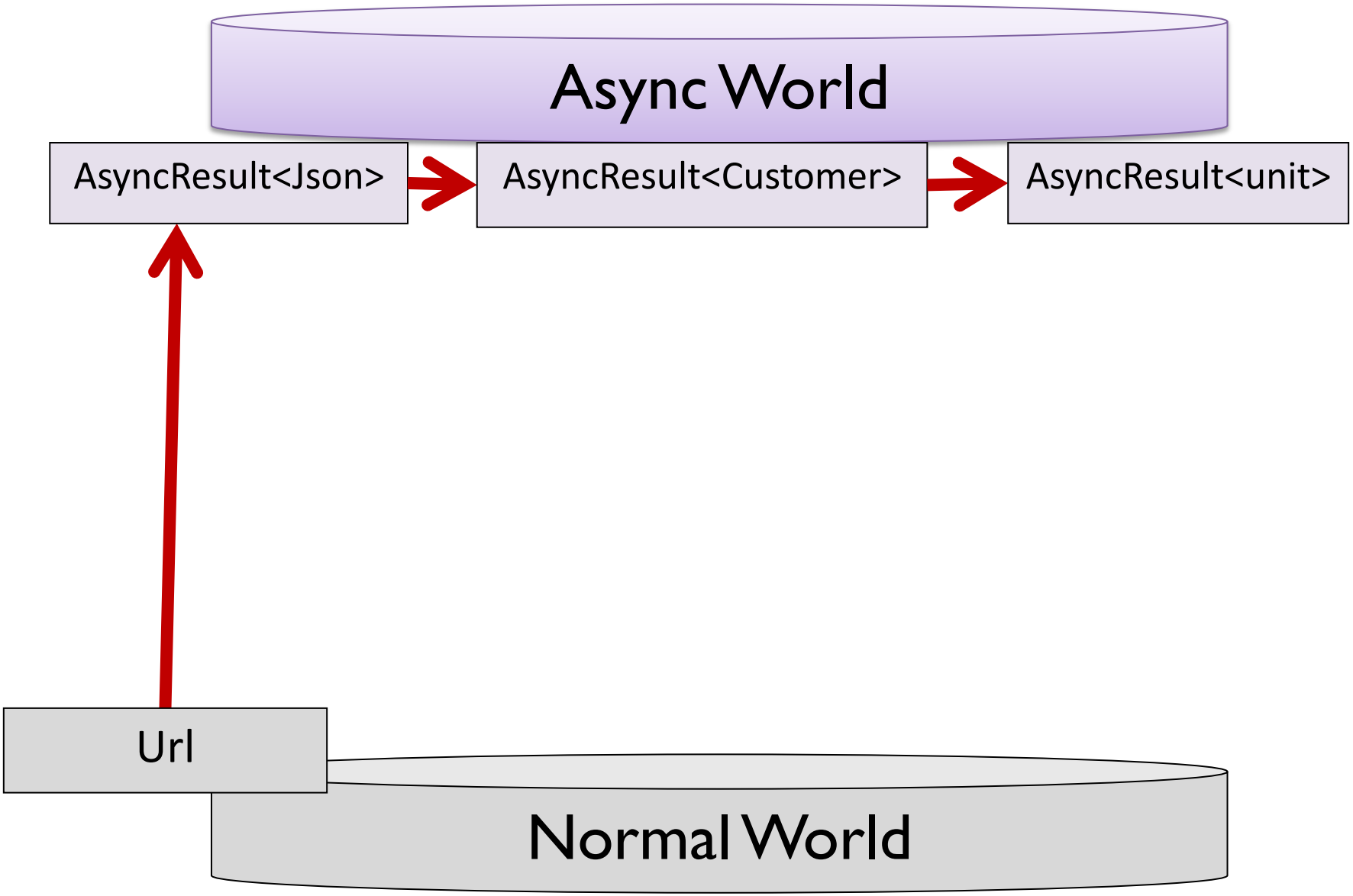
Store the customer



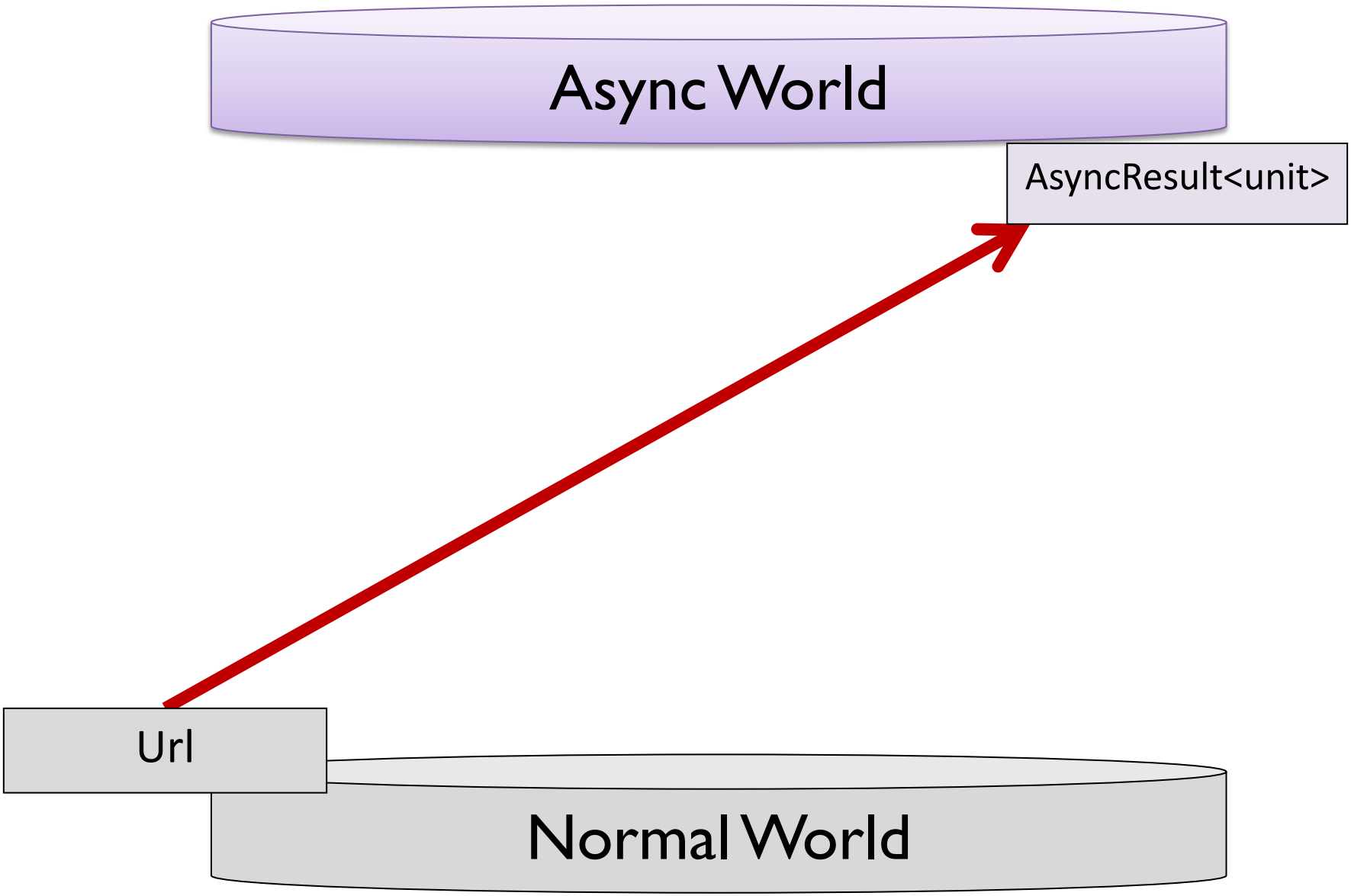
All steps are now composable



All steps are now composable



All steps are now composable into one single function



What the code looks like

```
let processCustomerDto jsonOrError =
  jsonOrError
  |> Result.bind decodeCustomerDto
  |> Result.bind createValidCustomer

let downloadAndStoreCustomer url =
  url
  |> downloadFile
  |> Async.map processCustomerDto
  |> AsyncResult.bind storeCustomer
```

It takes much longer to explain it than to write it!

In conclusion…

- FP jargon is not that scary
  - Can you see why monads are useful?
- The FP toolkit is very generic
  - FP's use these core functions constantly!
- You can now recognize "map", "apply" and "bind"
  - Don’t expect to understand them all straight away.

End