The Functional Programming Toolkit

(NDC London 2019)

@ScottWlaschin

fsharpforfunandprofit.com

Why do functional programmers use so many strange words?

Functional programming is scary

Functor

Currying

Catamorphism

Applicative

Monad

Monoid

Functional programming is unfamiliar ~~scary~~

"Mappable"

Currying

"Collapsables"

Applicative

"Chainable"

"Aggregatable"

Object oriented programming is scary

This!

Generics

Interface

Polymorphism

Inheritance

SRP, OCP, LSP, ISP, DIP, Oh noes..

SOLID

Covariance

...don't forget IoC, DI, ABC, MVC, etc., etc...

Functor

Monoid

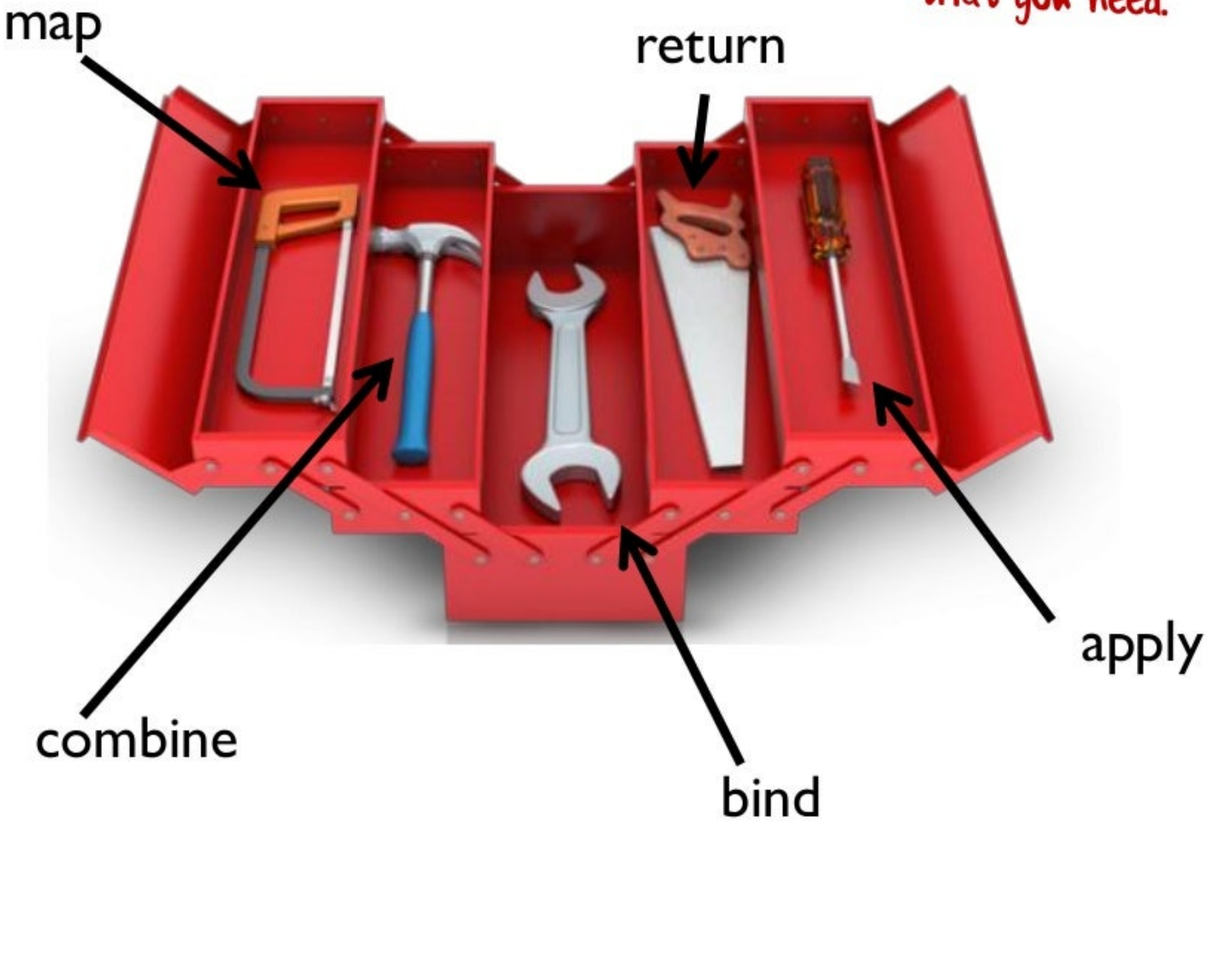
These aren't just academic concepts.
They are actually useful tools!

Applicative

Monad

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`

This talk

A whirlwind tour of many sights

Don't worry if you don't understand everything

Core principles of statically-typed FP

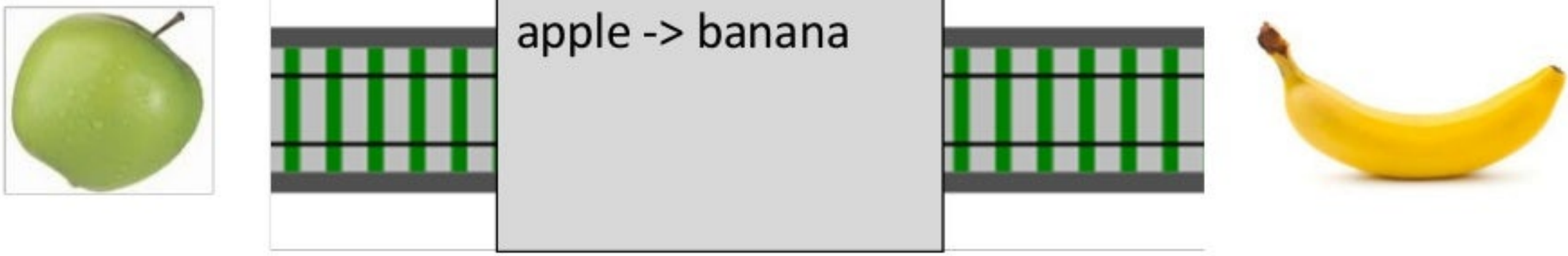
Core principles of (statically-typed) FP

Functions are things

Composition everywhere

Core FP principle:

Functions are things



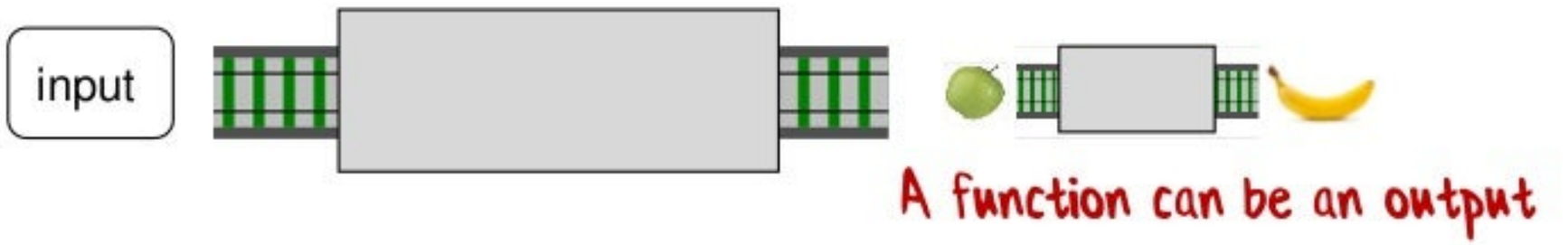
A function is a thing which transforms inputs to outputs

Another word for reusable!

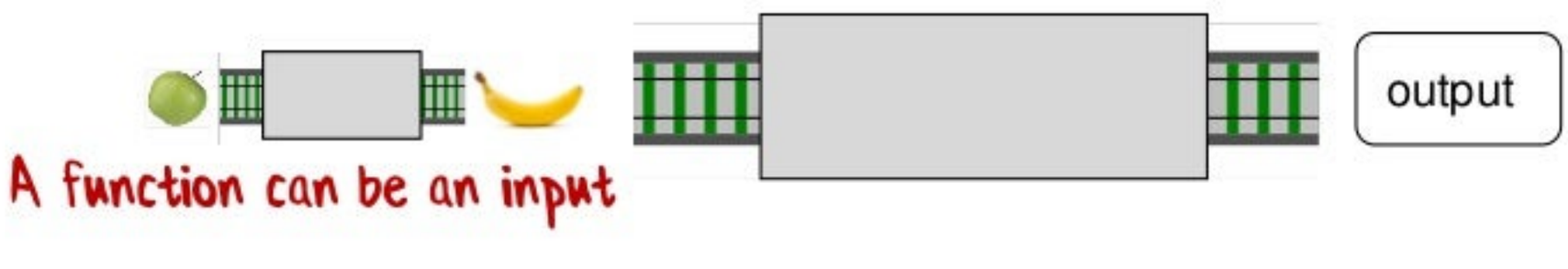
A function is a standalone thing, not attached to a class

It can be used for inputs and outputs of other functions

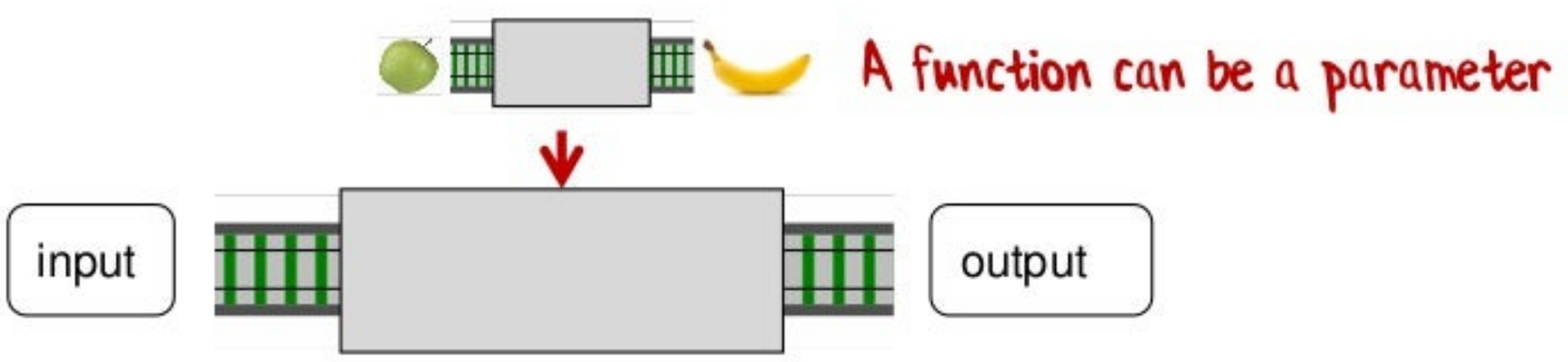
A function can be an output



A function can be an input



A function can be a parameter



You can build very complex systems
from this simple foundation!

Most of the tools in the functional toolbox are "function transformers" that convert functions to functions

Core FP principle:

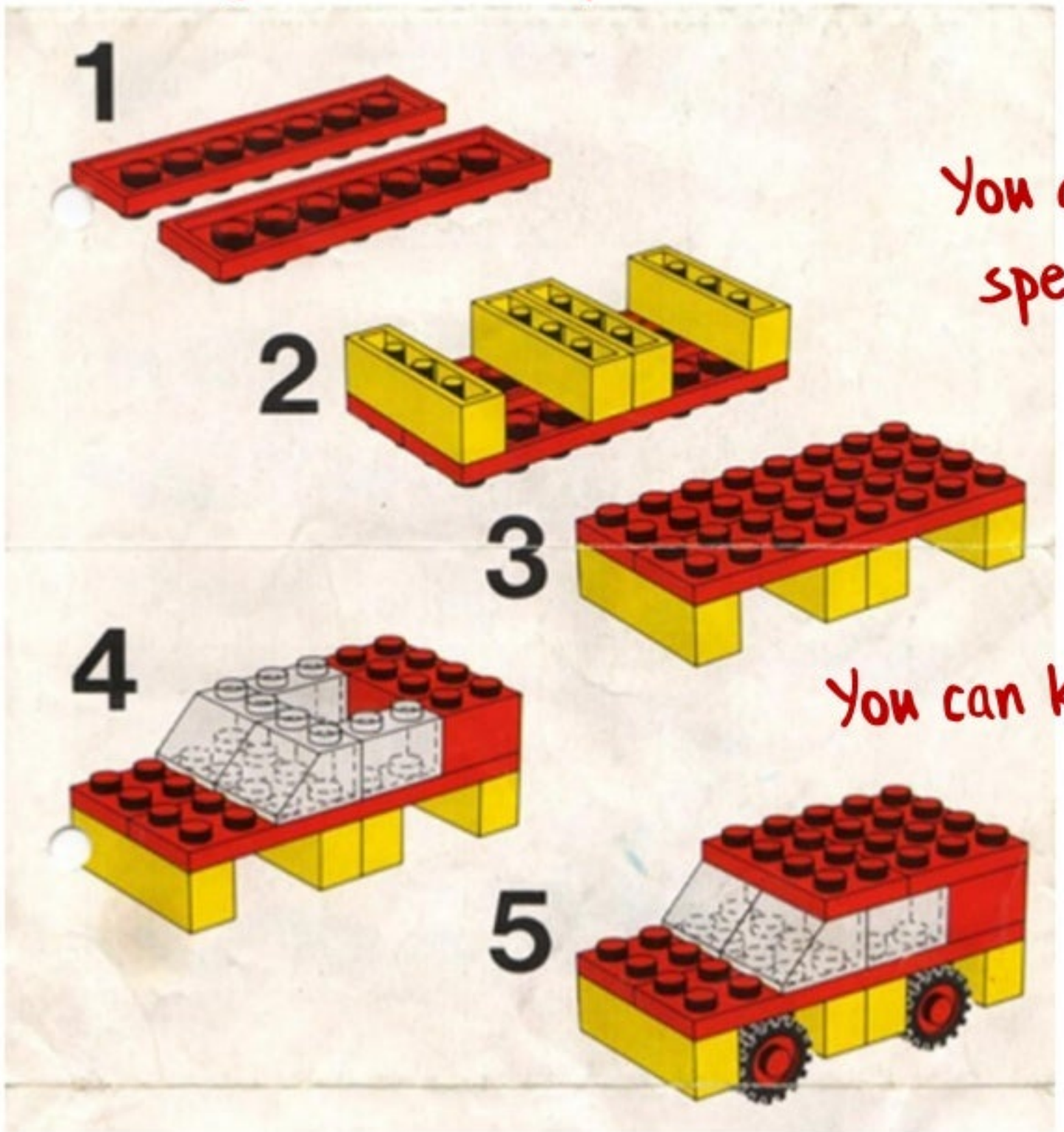
Composition everywhere

Lego Philosophy

1. All pieces are designed to be connected
2. Connect two pieces together and get another "piece" that can still be connected
3. The pieces are reusable in many contexts

All pieces are designed to be connected

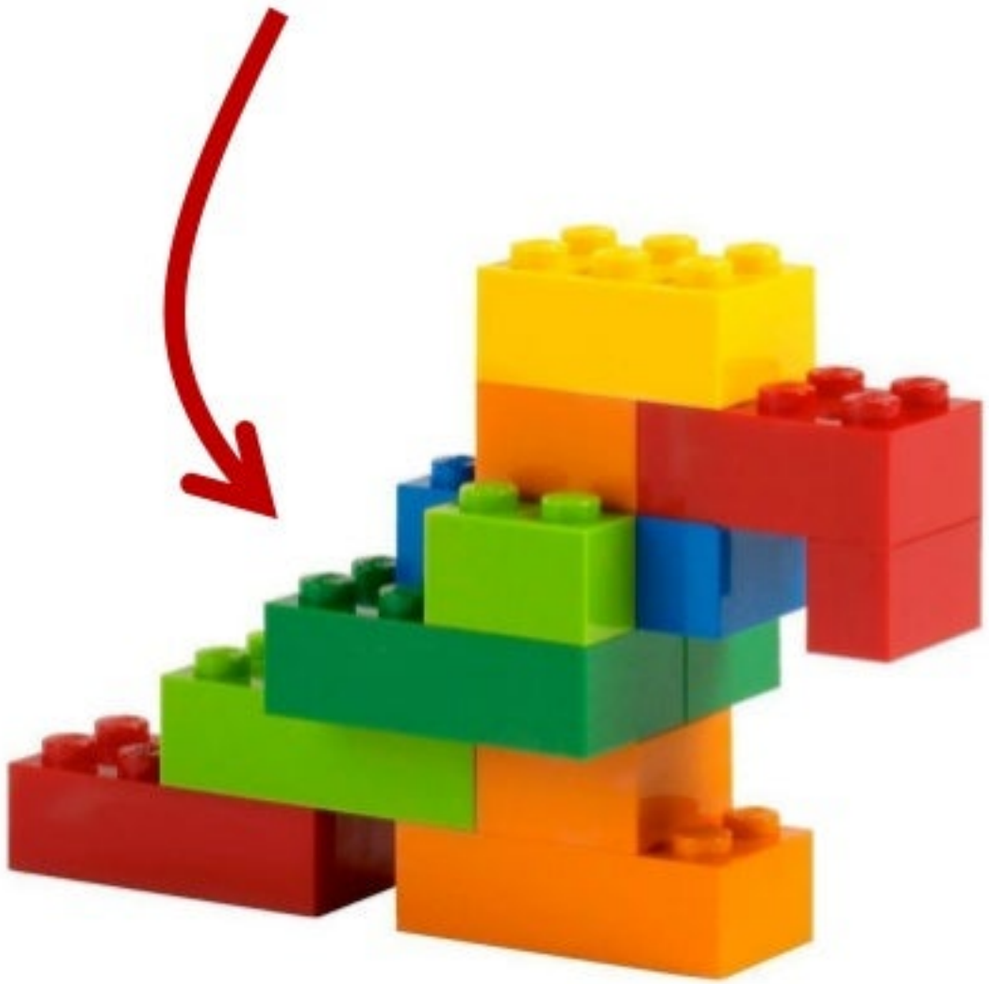
Connect two pieces together and
get another "piece" that can still be connected

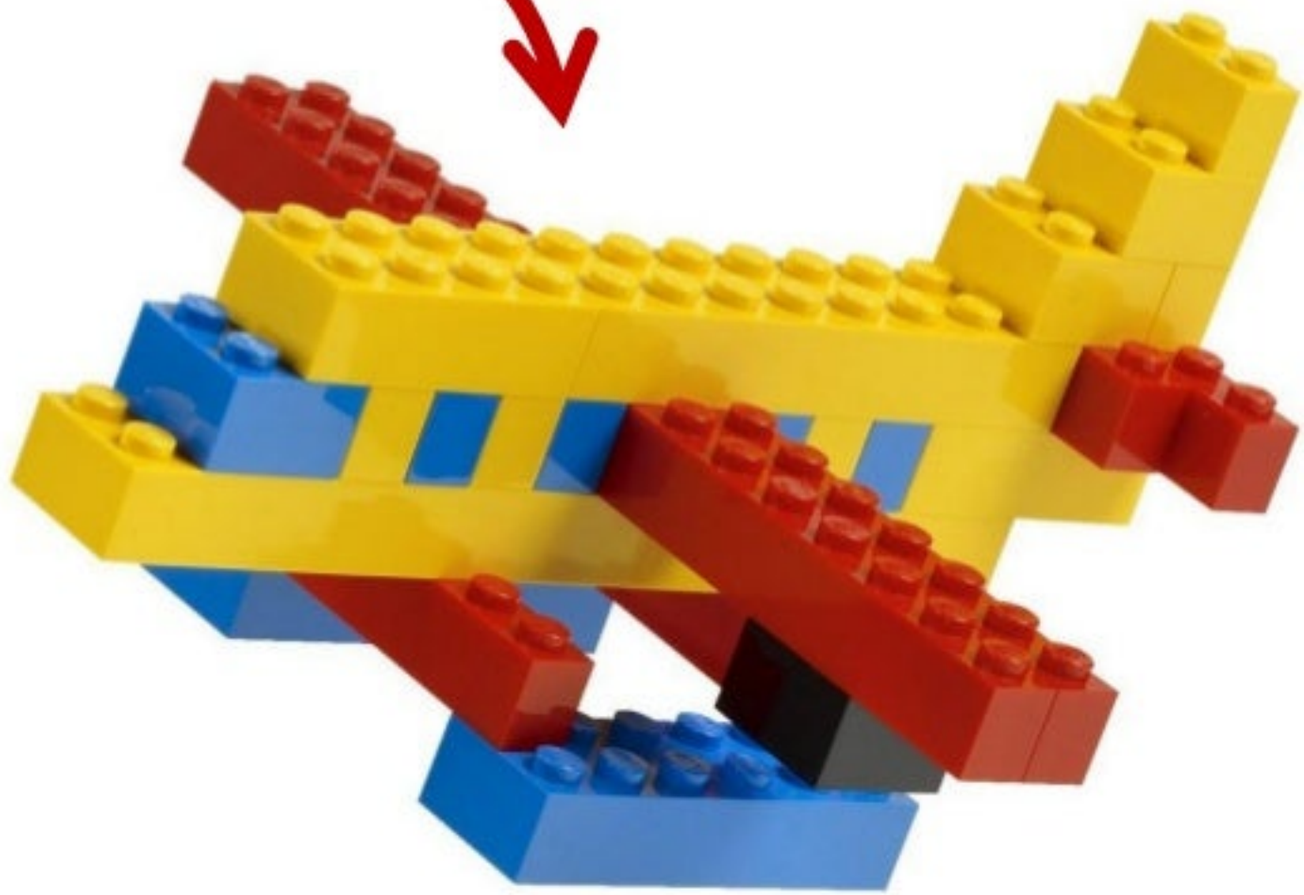


You don't need to create a special adapter to make connections.

You can keep adding and adding.

The pieces are reusable in different contexts

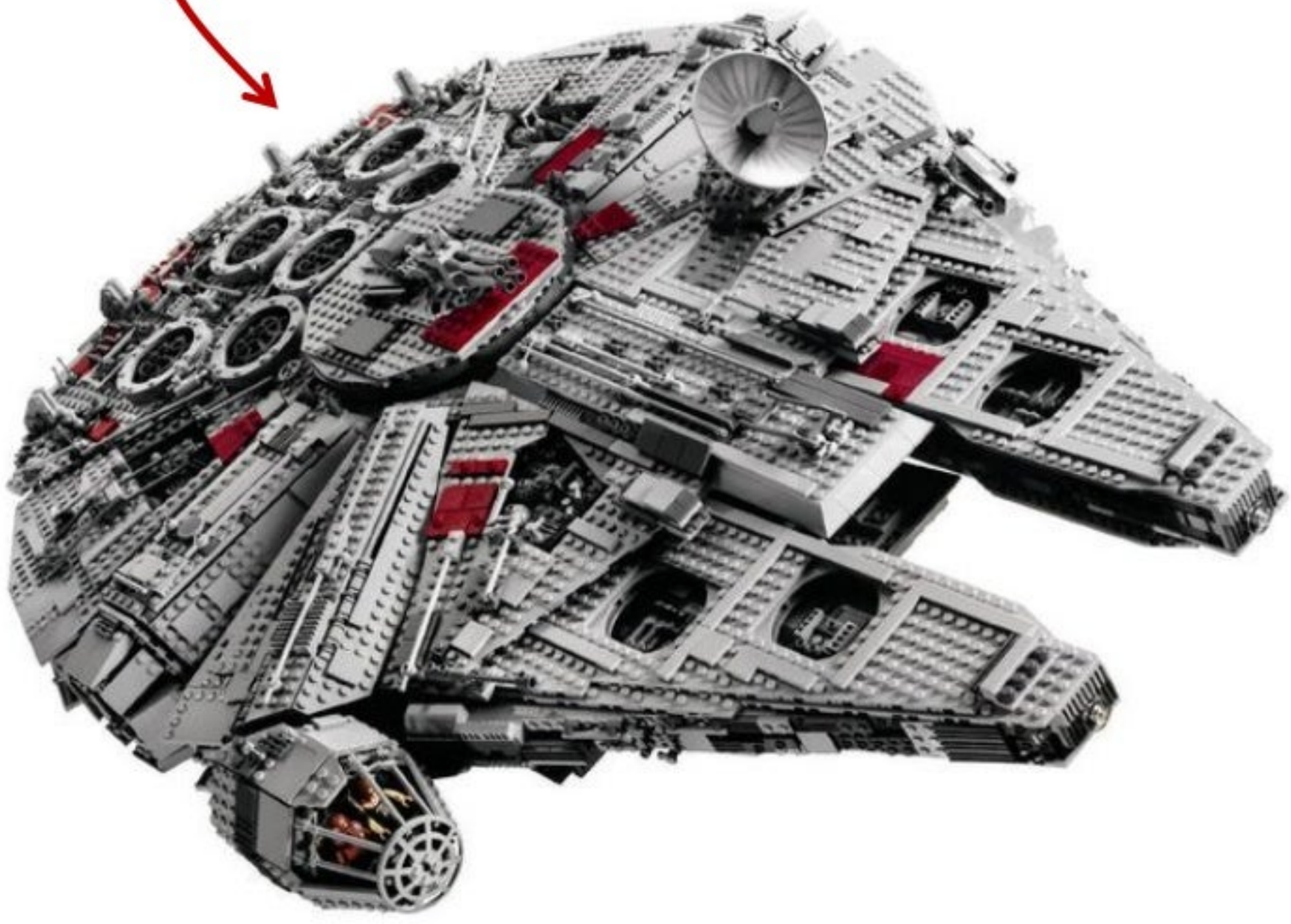




They are self contained.
No strings attached (literally).

Make big things from small things in the same way

The Power of Composition



Statically typed

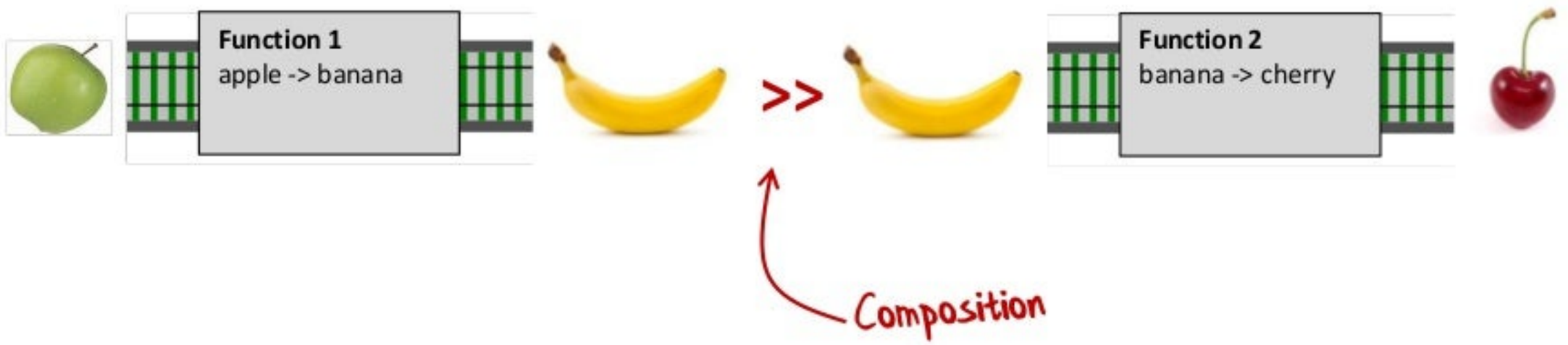
Functional Programming Philosophy

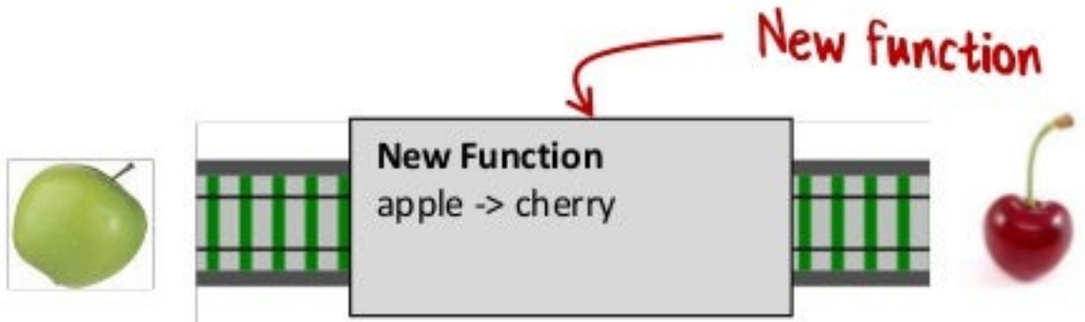
- Design functions that do one thing well
  - Functions can be reused in different contexts
- Design functions to work together
  - Expect the output of every function to become the input to another, as yet unknown, function
- Use types to ensure that inputs match outputs

The functional toolkit

Function composition


Function 1
apple -> banana
Function 2
banana -> cherry


Function 1
apple -> banana
>>
Function 2
banana -> cherry
Composition

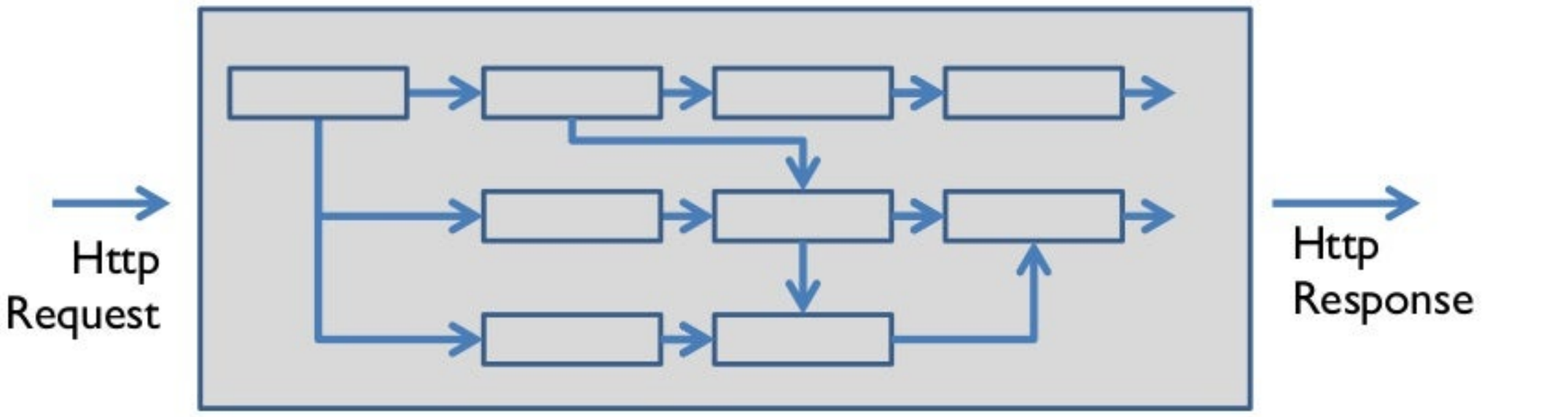


Can't tell it was built from smaller functions!

Where did the banana go?

A web application built from functions only (no classes!)

A web application built from functions only (no classes!)



The Power of Composition

I have a whole talk on "The Power of Composition" at
fsharpforfunandprofit.com/composition

Tool #1

Combining things with Monoids

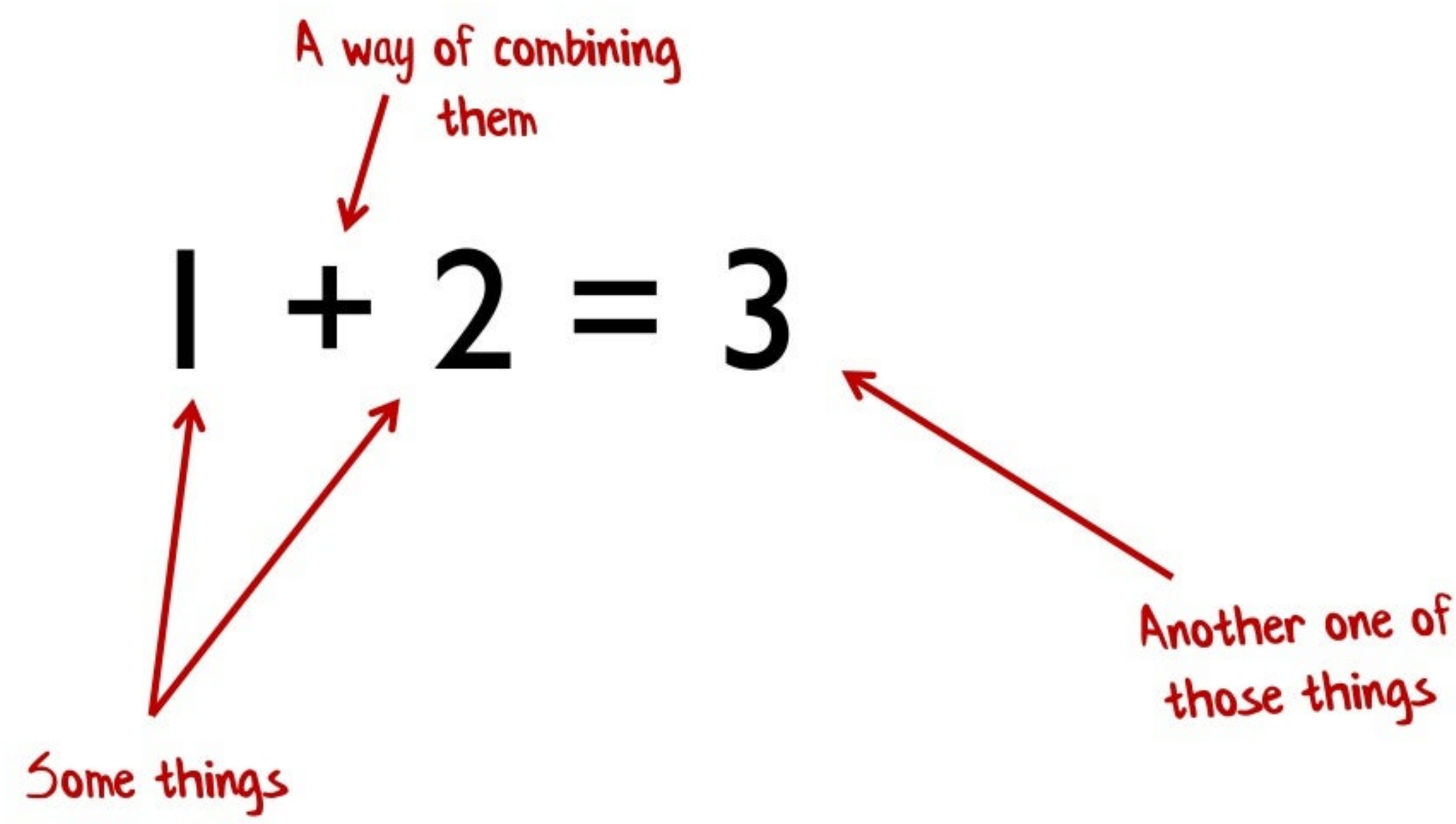

WARNING
Mathematics
Ahead

$$1 + 2 = 3$$

$$1 + (2 + 3) = (1 + 2) + 3$$

$$1 + 0 = 1$$
$$0 + 1 = 1$$

Now, let's think like a mathematician


A way of combining them
1 + 2 = 3
Some things
Another one of those things

A way of combining them

$$2 \times 3 = 6$$

Some things

Another one of those things

A way of combining them

"a" + "b" = "ab"

Some things

Another one of those things

A way of combining them

concat([a],[b]) = [a; b]

Some things

Another one of those things

Is an integer

$$1 + 2$$

Is an integer

$$1 + 2 + 3$$

$$1 + 2 + 3 + 4$$

A pairwise operation has become an operation that works on lists!

Order of combining doesn't matter

$$1 + (2 + 3) = (1 + 2) + 3$$

$$1 + 2 + 3 + 4$$

$$(1 + 2) + (3 + 4)$$

$$((1 + 2) + 3) + 4$$

All the same

Order of combining does matter

$$1 - (2 - 3) = (1 - 2) - 3$$

1 + 0 = 1

0 + 1 = 1

A special kind of thing that when you combine it with something, just gives you back the original something

$$42 * 1 = 42$$

$$1 * 42 = 42$$

A special kind of thing that when you combine it with something, just gives you back the original something

"" + "hello" = "hello"

"hello" + "" = "hello"

"Zero" for strings

- You start with a bunch of things, *and* some way of combining them two at a time.

- **Rule 1 (Closure)**: The result of combining two things is always another one of the things.

- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.

- **Rule 3 (Identity element)**: There is a special thing called "zero" such that when you combine any thing with "zero" you get the original thing back.

A monoid!

Hey, this is just the kind of jargon that makes my head hurt!

Show me why this is useful!

- **Rule 1 (Closure)**: The result of combining two things is always another one of the things.
- **Benefit**: converts pairwise operations into operations that work on lists.

Pairwise operations can be converted into operations that work on lists.

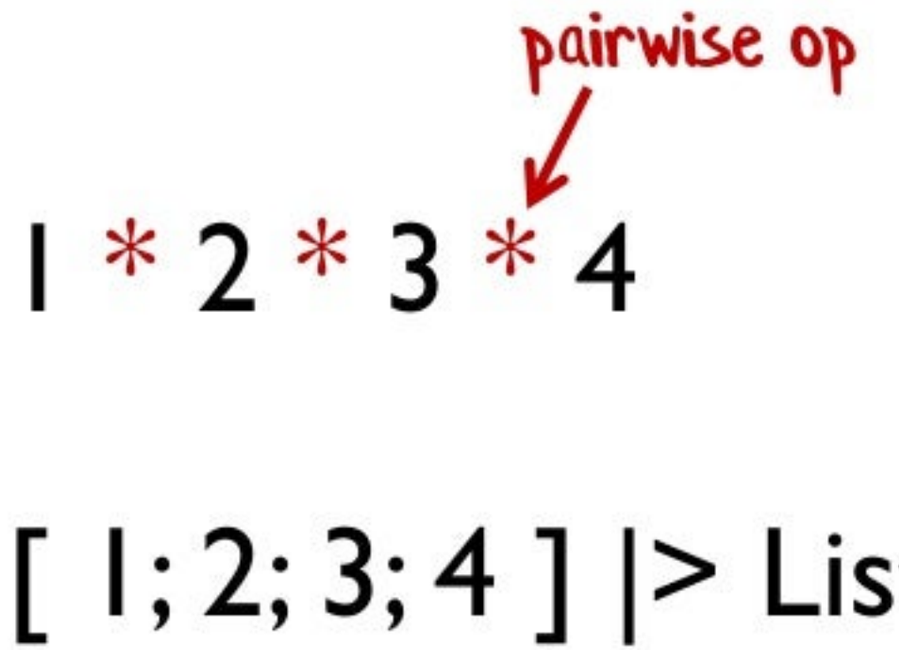
pairwise op

```
1 + 2 + 3 + 4
```

```
[ 1; 2; 3; 4 ] |> List.reduce (+)
```

We'll see this a lot!

Pairwise operations can be converted into operations that work on lists.



Pairwise operations can be converted into operations that work on lists.

pairwise op

```
"a" + "b" + "c" + "d"
```

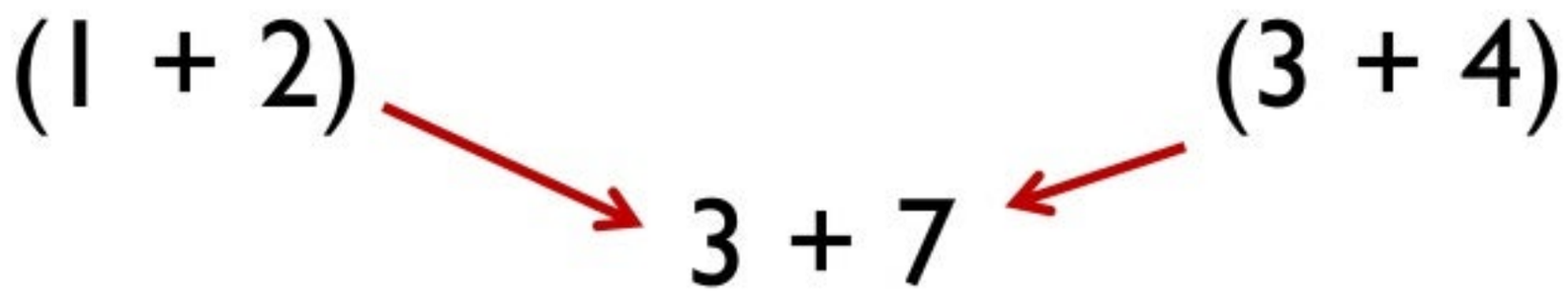
```
[ "a"; "b"; "c"; "d" ] |> List.reduce (+)
```

Benefit of Associativity: Divide and conquer, parallelization, and incremental accumulation.

Parallelization:

1 + 2 + 3 + 4

Parallelization:



Benefit of Associativity: Divide and conquer, parallelization, and incremental accumulation.

Benefit of Associativity: Divide and conquer, parallelization, and incremental accumulation.

(1 + 2 + 3)

Benefit of Associativity: Divide and conquer, parallelization, and incremental accumulation.

(1 + 2 + 3) + 4

Benefit of Associativity: Divide and conquer, parallelization, and incremental accumulation.

(6) + 4

- How can I use reduce on an empty list?
- In a divide and conquer algorithm, what should I do if one of the "divide" steps has nothing in it?
- When using an incremental algorithm, what value should I start with when I have no data?

Identity element

- **Benefit:** Initial value for empty or missing data

If zero is missing, it is called a semigroup

Tip:

Simplify aggregation code with monoids

```
type OrderLine = {Qty:int; Total:float}

let orderLines = [
    {Qty=2; Total=19.98}
    {Qty=1; Total= 1.99}
    {Qty=3; Total= 3.99} ]
```

```
type OrderLine = {Qty:int; Total:float}
```

Any combination of monoids is also a monoid

```
let combine line1 line2 =
    let newQty = line1.Qty + line2.Qty
    let newTotal = line1.Total + line2.Total
    {Qty=newQty; Total=newTotal}
```

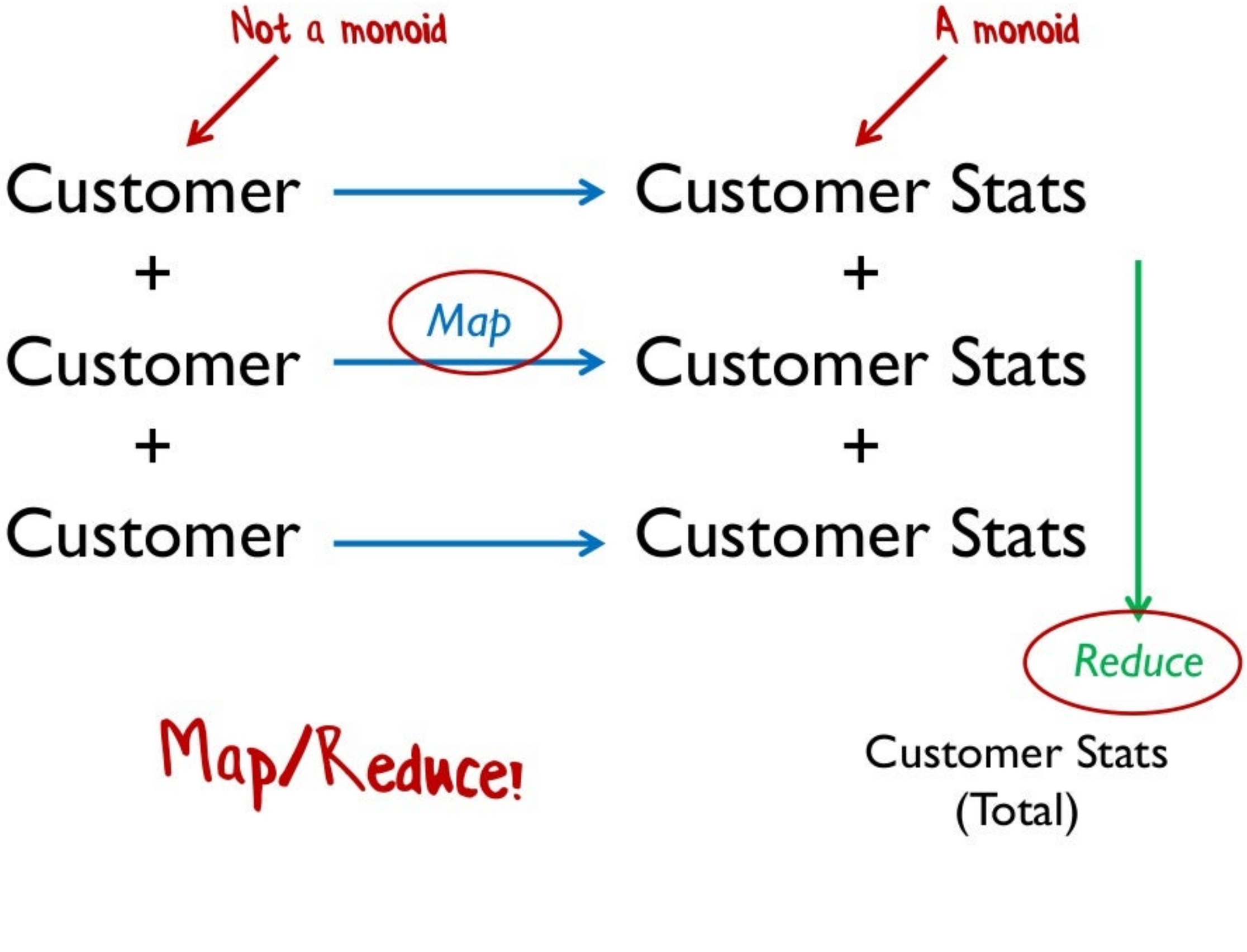
Write a pairwise combiner

```
orderLines |> List.reduce combine
                                    // {Qty=6; Total= 25.96}
```

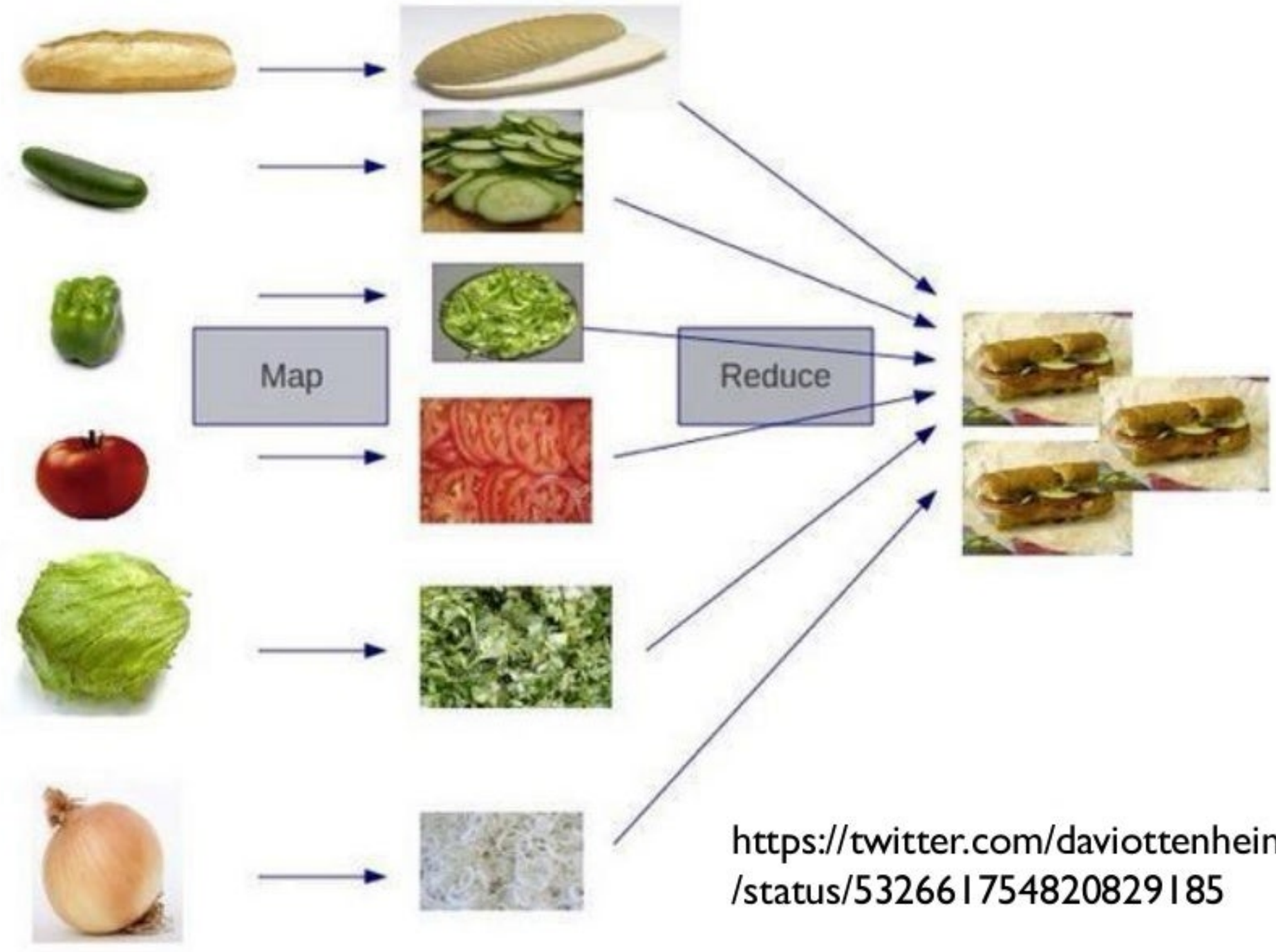
Profit!

Pattern:

Convert non-monoids to monoids


Not a monoid
A monoid
Customer
+
Customer
+
Customer
Customer Stats
+
Customer Stats
+
Customer Stats
Map
Reduce
Customer Stats
(Total)
Map/Reduce!

Hadoop make me a sandwich



https://twitter.com/daviottenheimer/status/532661754820829185

Beware: You will soon start seeing monoids everywhere

~~Metrics guideline:~~
~~Use counters rather than rates~~

Alternative metrics guideline:

Make sure your metrics are monoids

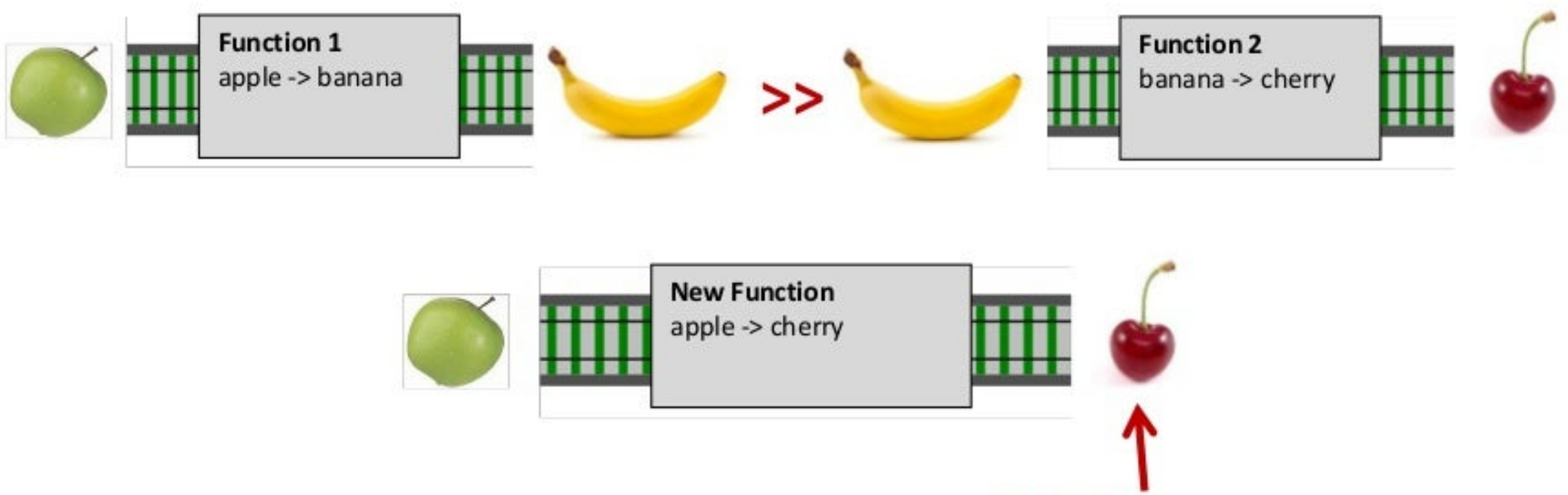
- incremental updates
- can handle missing data

Many "design patterns" are monoids

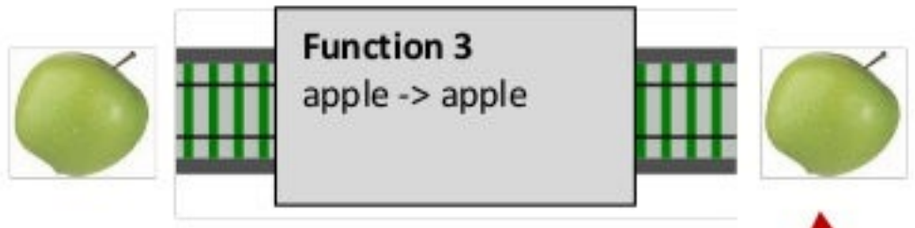
- Composite Pattern
- Null Object Pattern
- Composable commands
- Closure of operations (DDD)

Mark Seemann: http://blog.ploeh.dk/2019/01/28/better-abstractions-revisited/

Is function composition a monoid?


Function 1
apple -> banana
>>
Function 2
banana -> cherry
New Function
apple -> cherry
Not the same thing

Not a monoid ☹



Same thing

A monoid! ☺

Monoid summary

- A set of values and a combining function:
  `combine (aka concat, plus, <>, <+> )`
- Closed and associative and has a zero value
- Uses:
  - Reduce a list of values to a single value
  - Do parallel calculations
  - Do incremental calculations

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

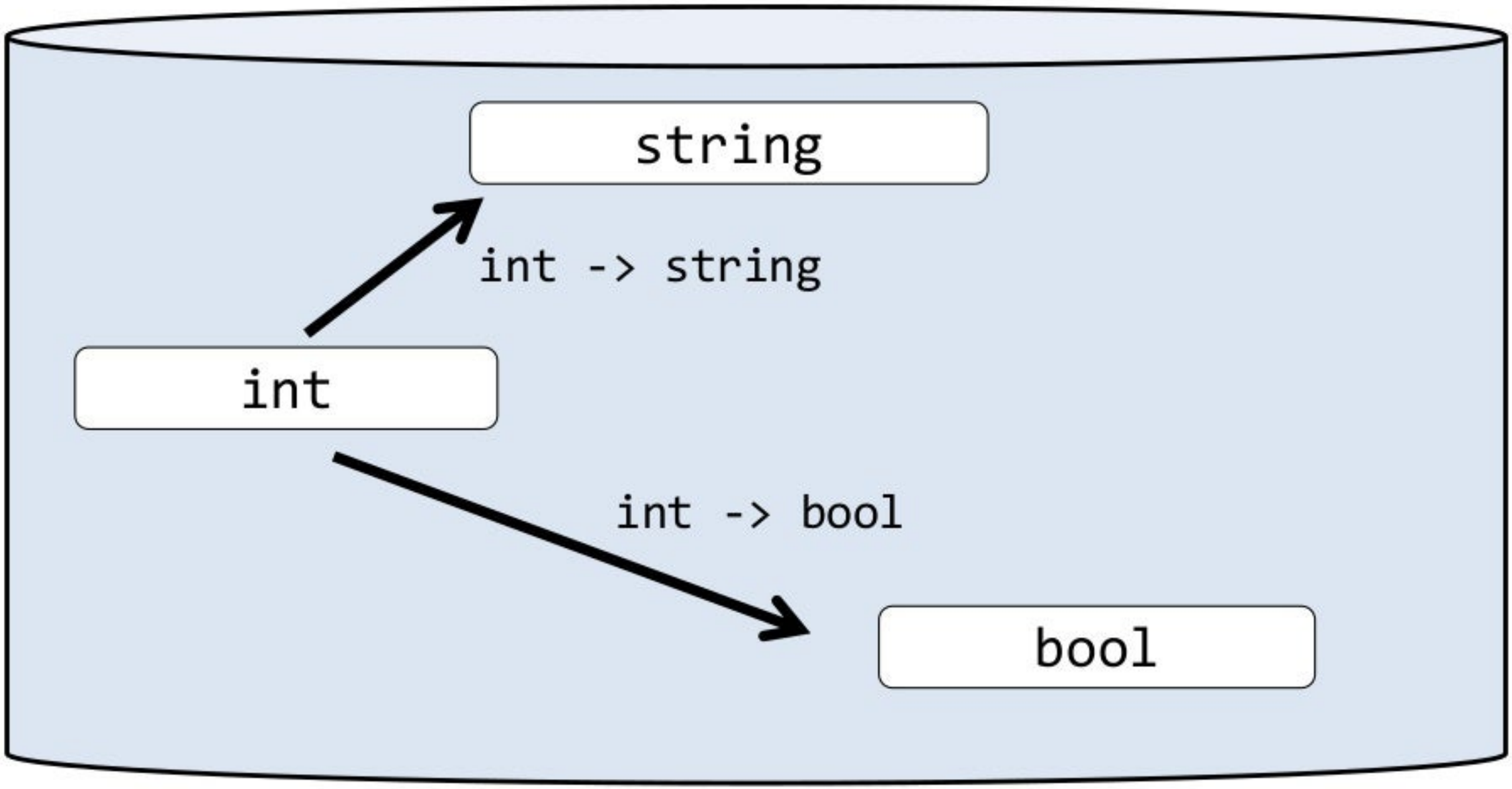
  `Async<_>`, `Task<_>`, `Random<_>`

- A type that carries state

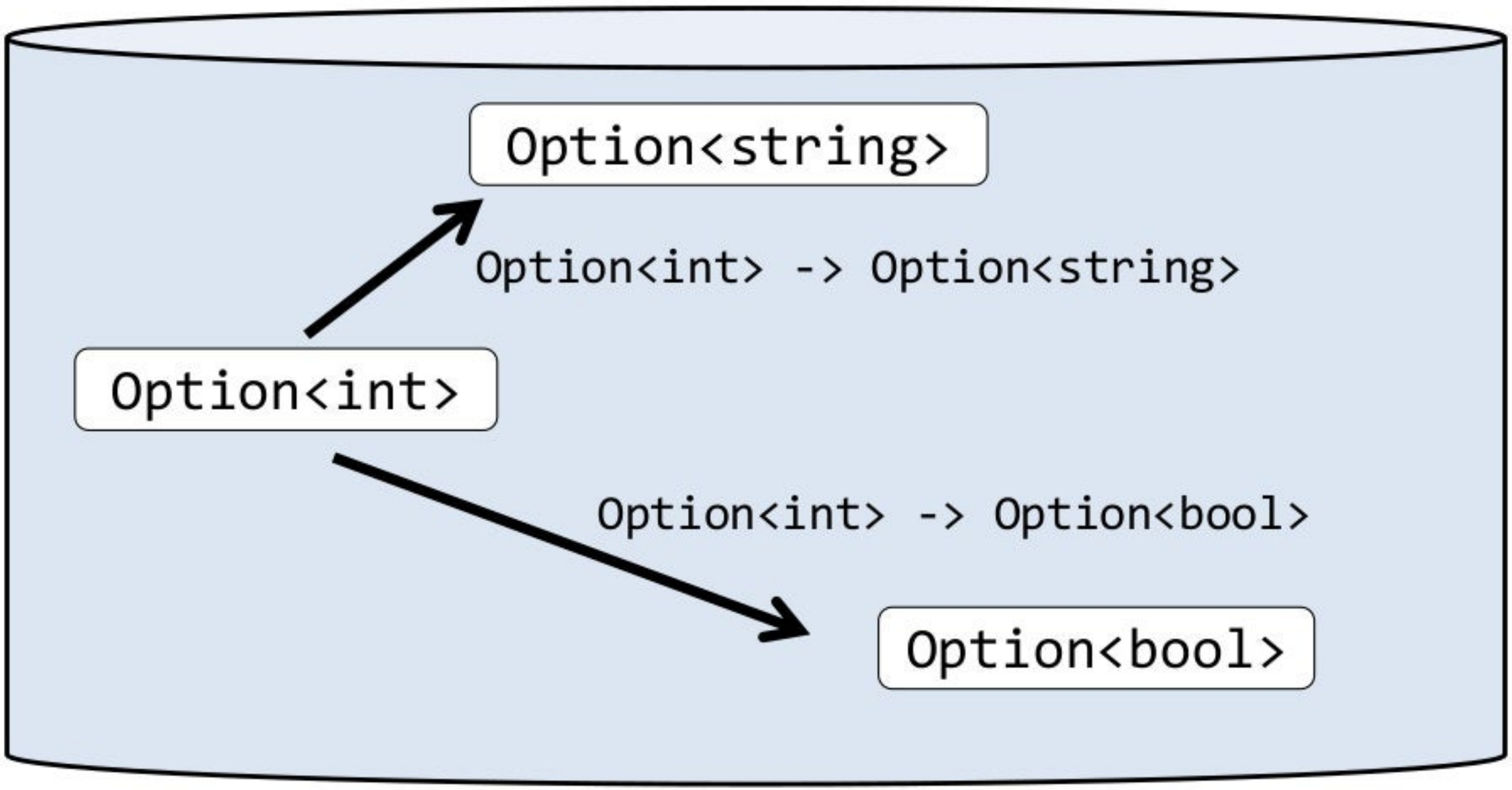
  `State<_>`, `Parser<_>`

"Normal" world vs. "Effects" world

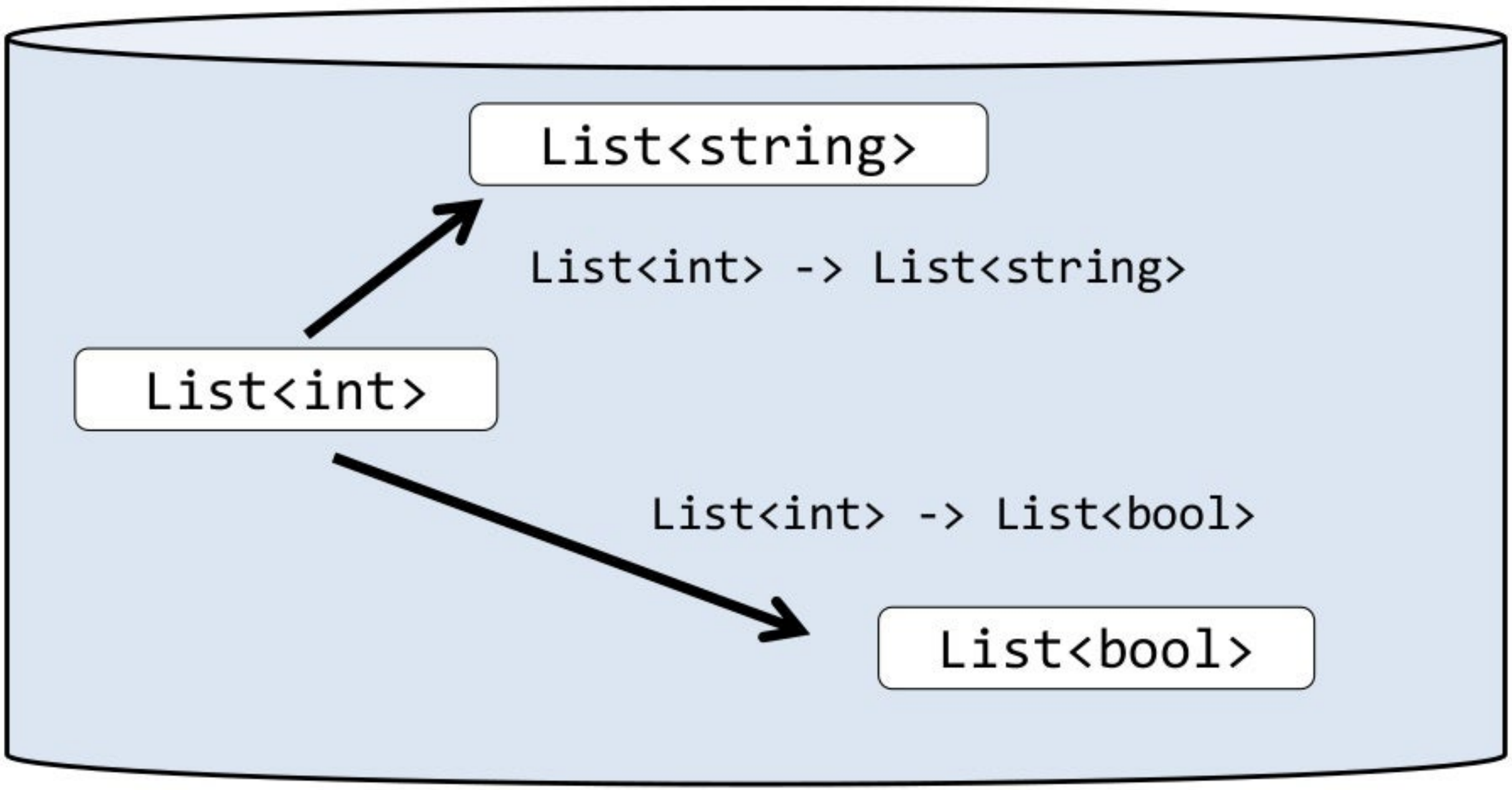
"Normal" world



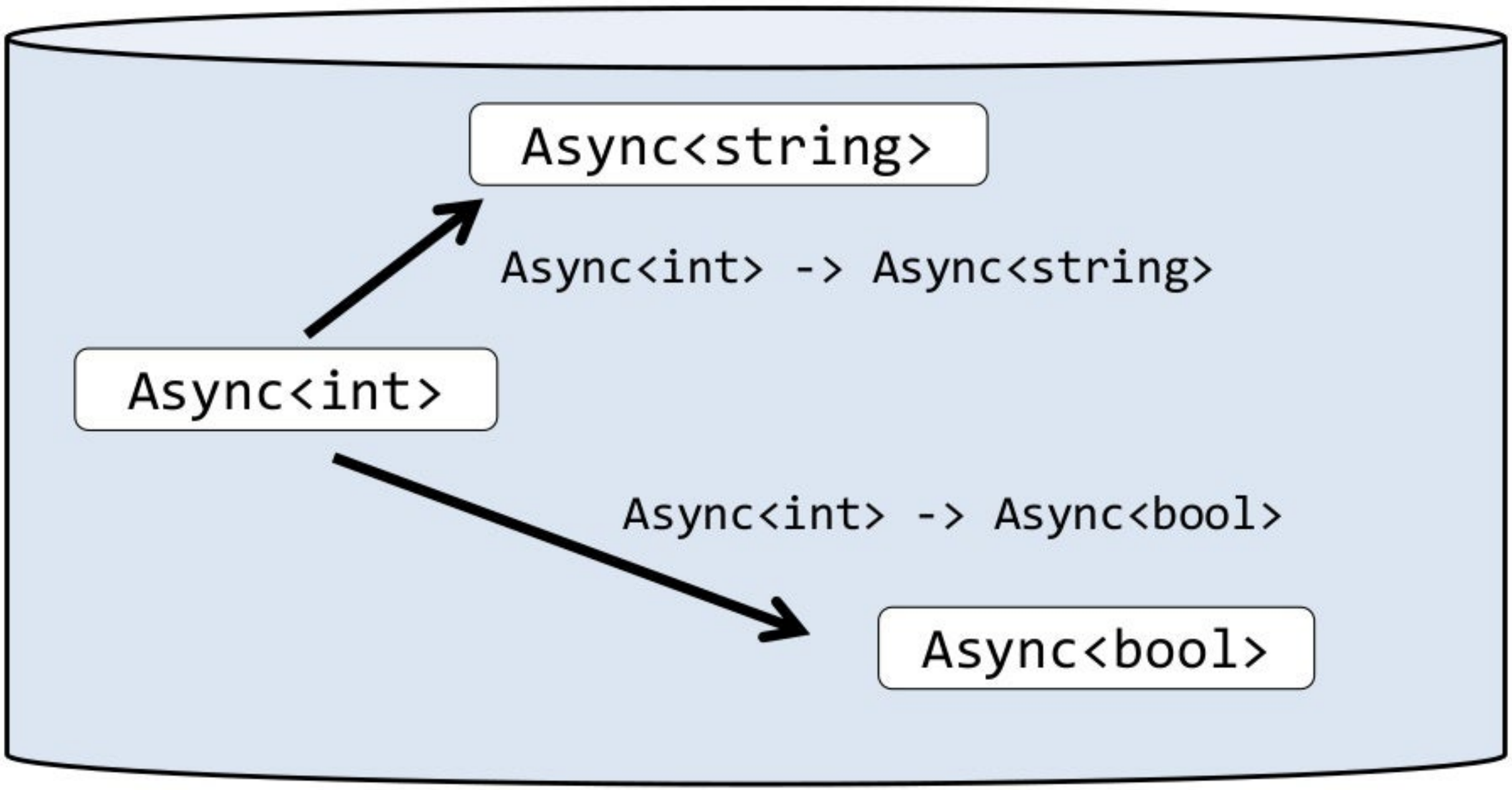
"Option" world



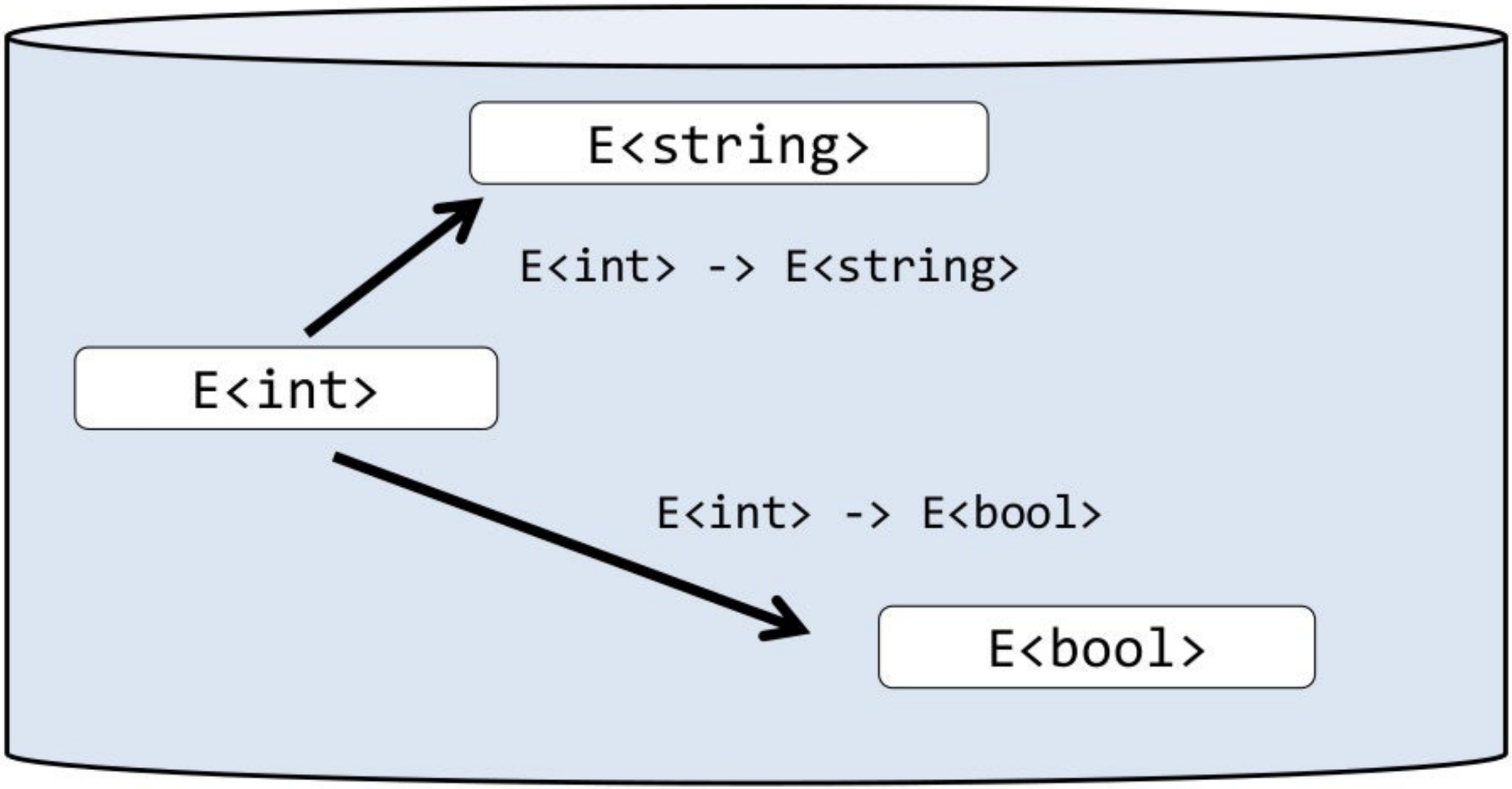
"List" world


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>

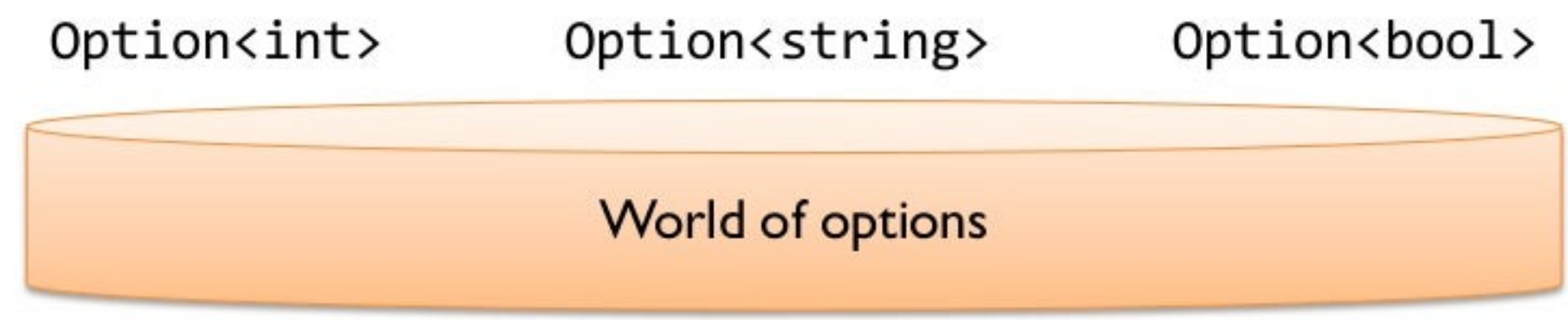
"Effects" world

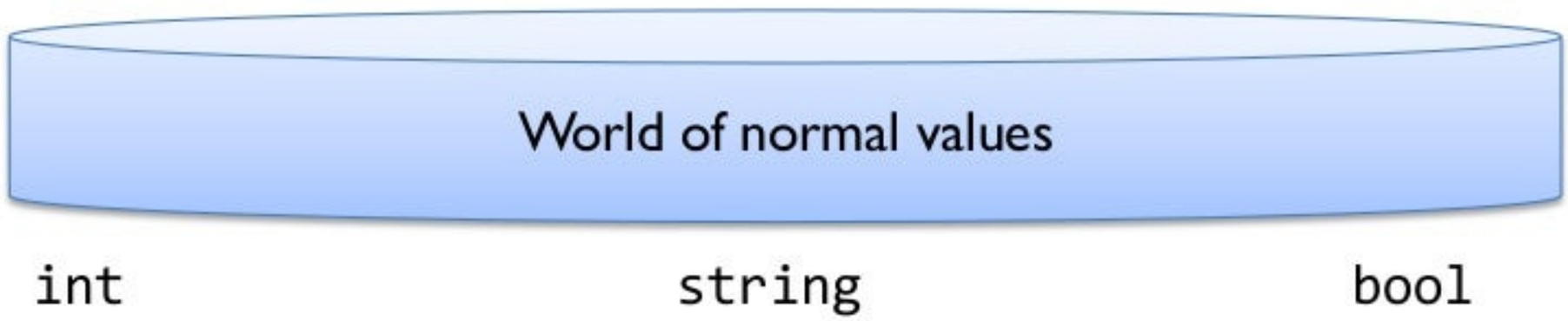


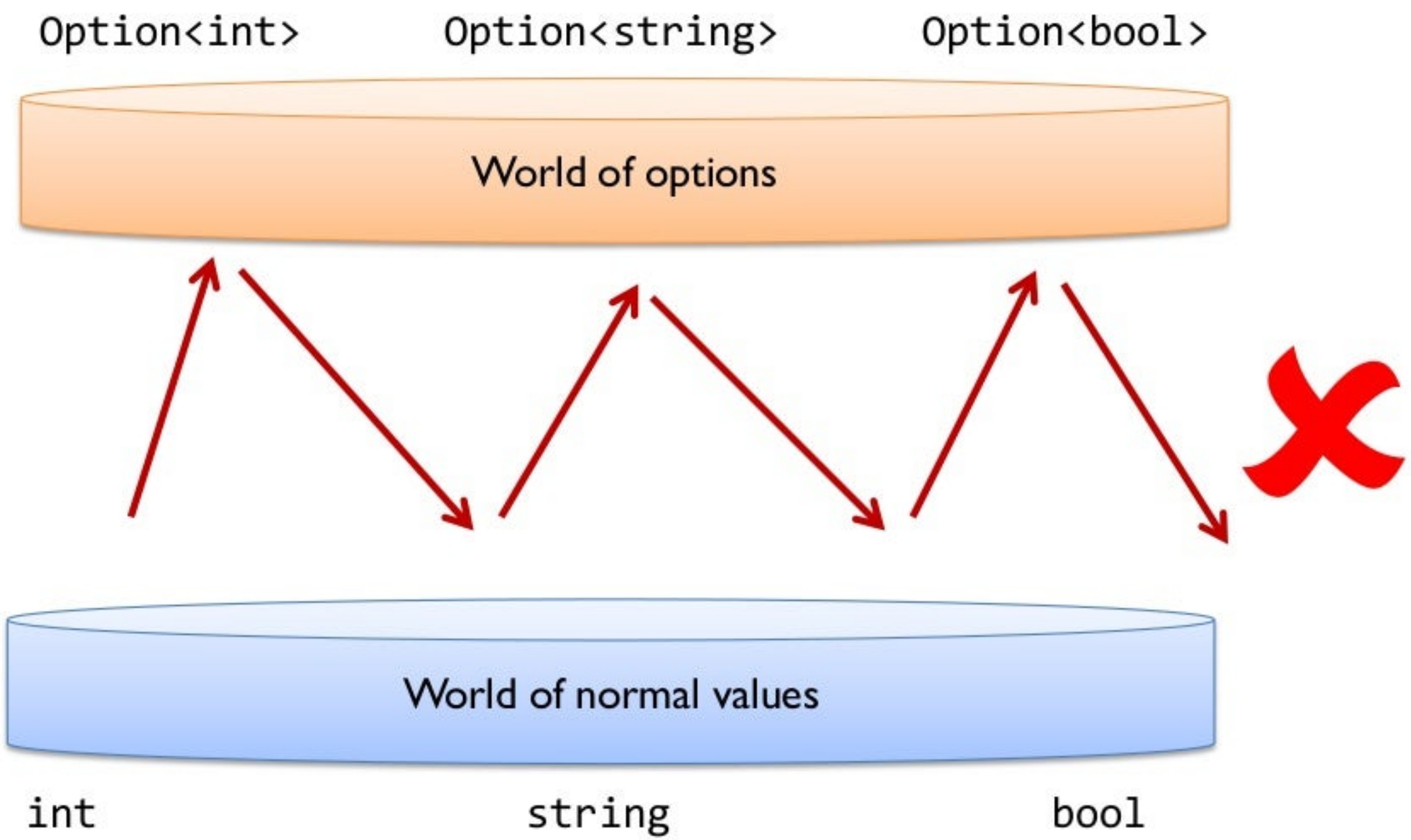
Problem:

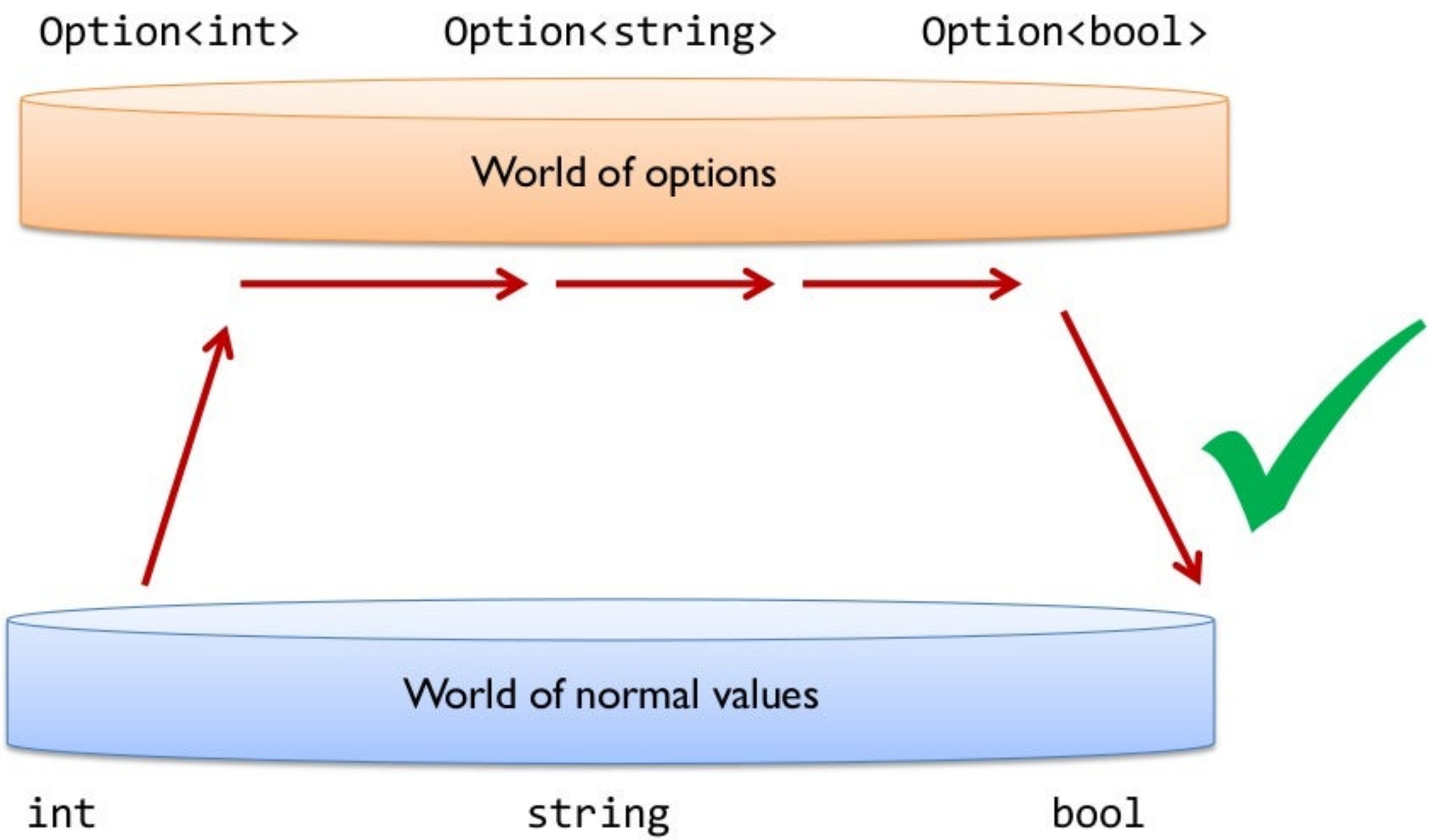
How to do stuff in an effects world?

Example:
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

```fsharp
let add42 x = x + 42

add42 1  // 43

add42 (Some 1)  // error
```

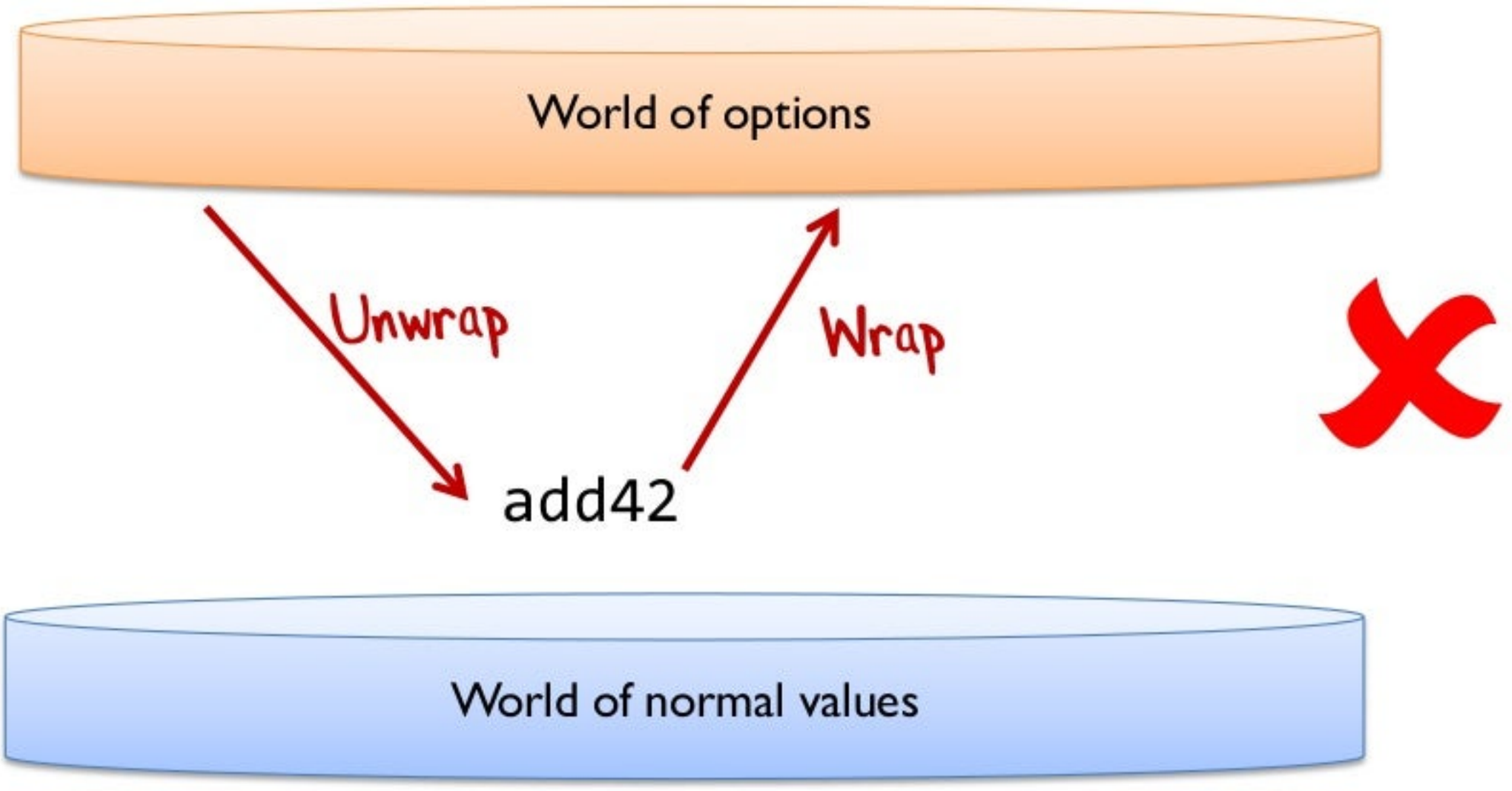
Only works on non-option values

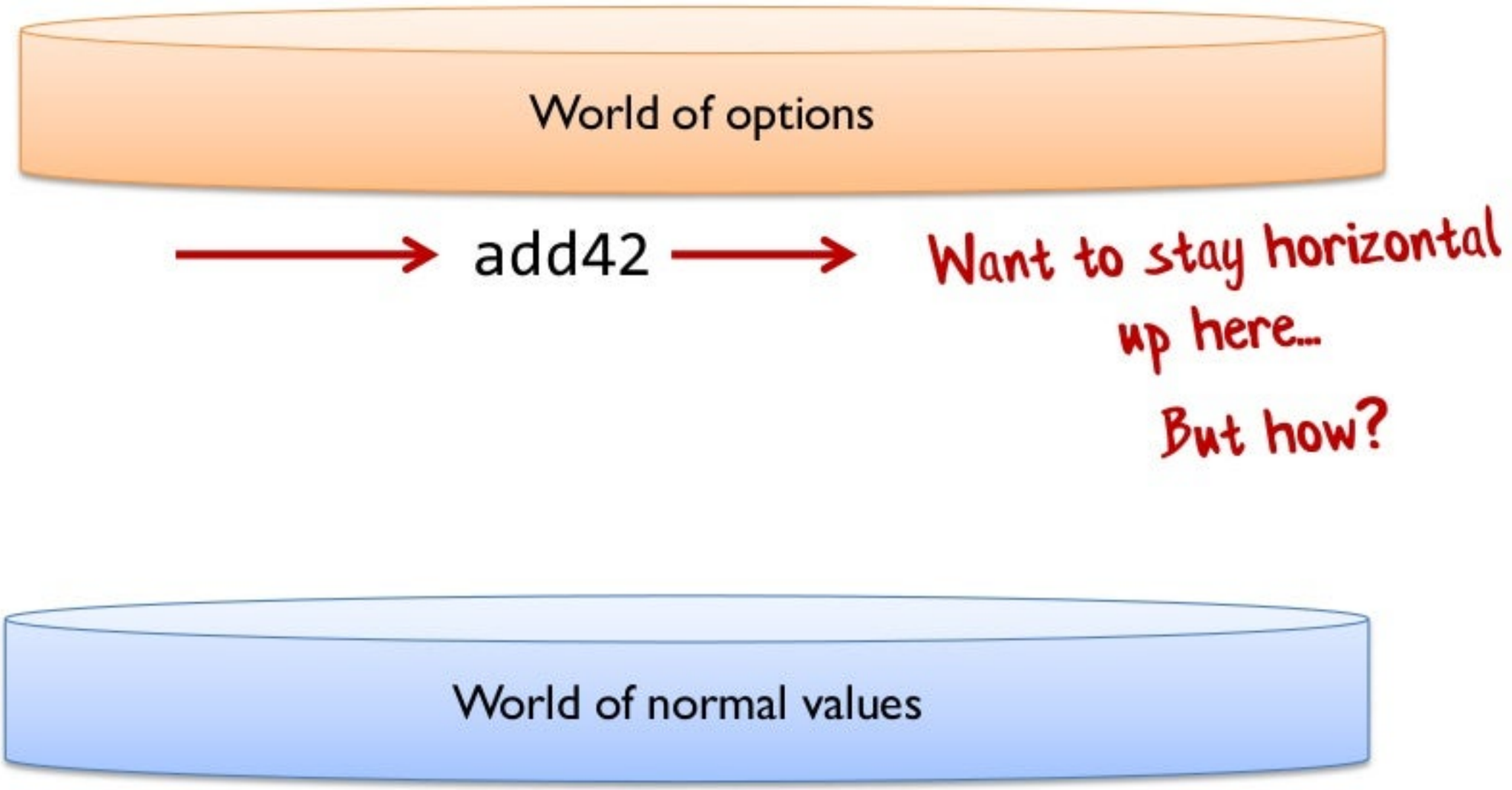
```
let add42ToOption opt =
    if opt.IsSome then
        let newVal = add42 opt.Value
        Some newVal
    else
        None
```

Unwrap

Apply

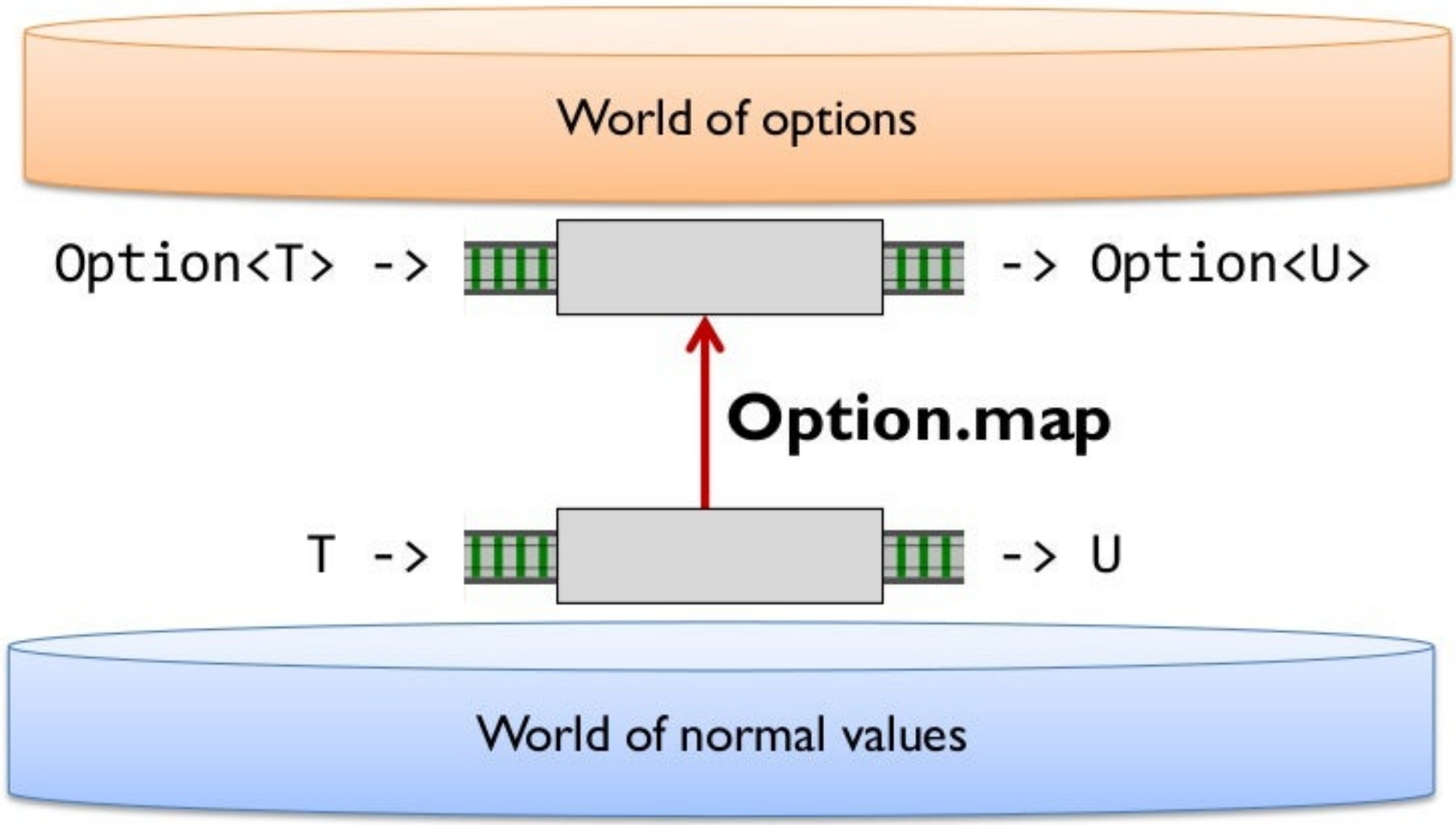
Wrap again


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 = ...
```

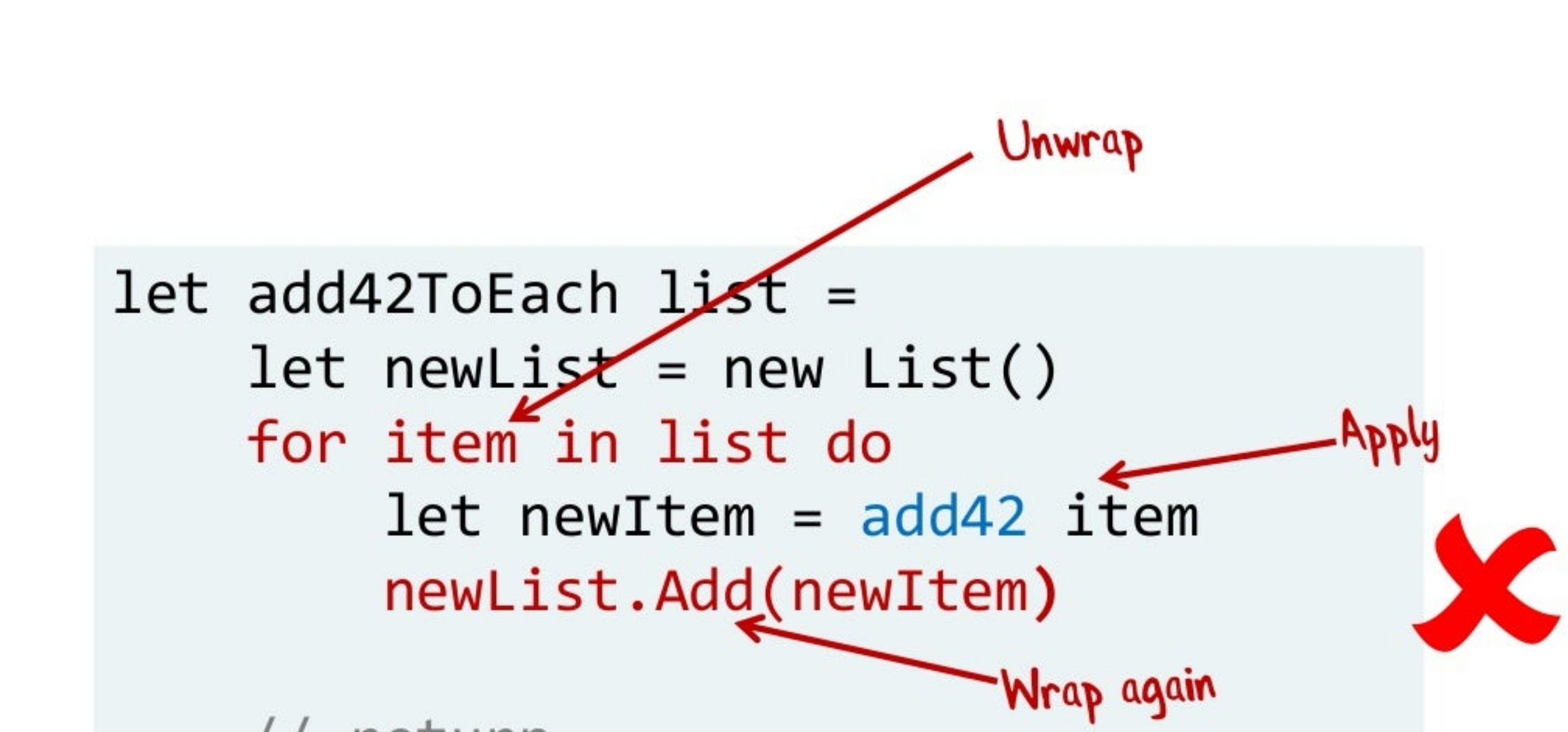
```
(Option.map add42) (Some 1)
```

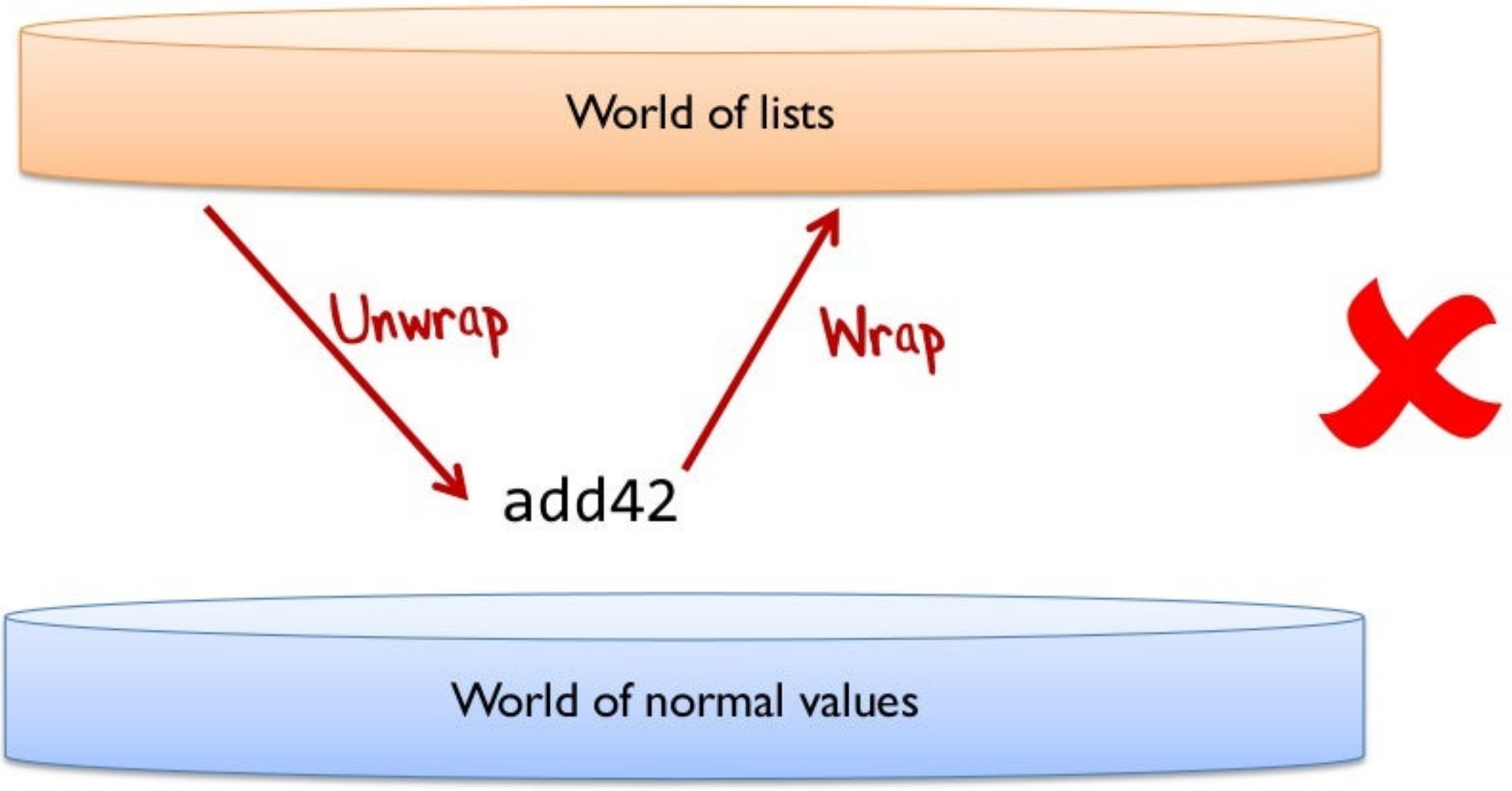
Normally just use inline

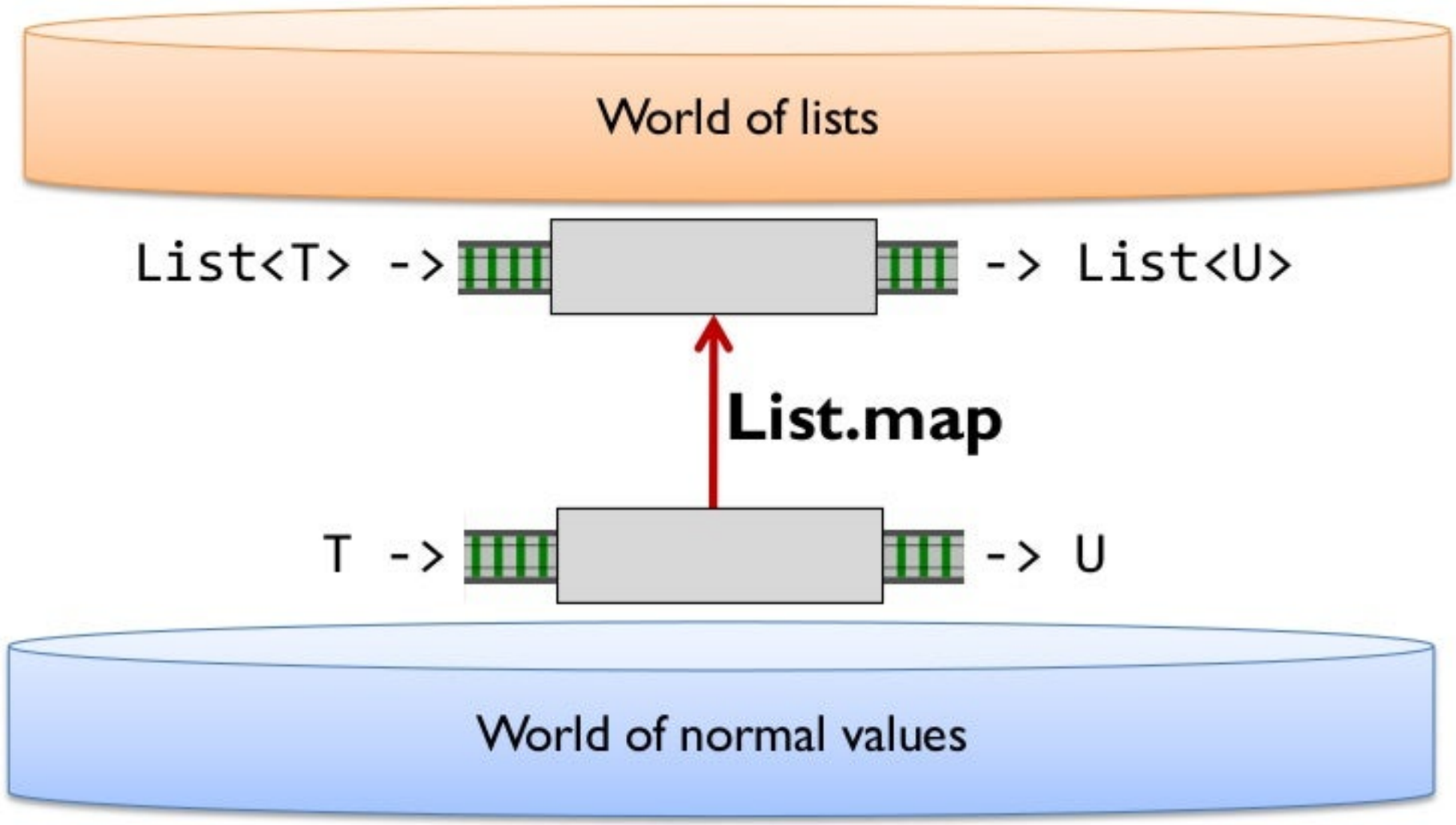
Example:
Working with List world

```
let add42ToEach list =
    let newList = new List()
    for item in list do
        let newItem = add42 item
        newList.Add(newItem)

    // return
    newList
```




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

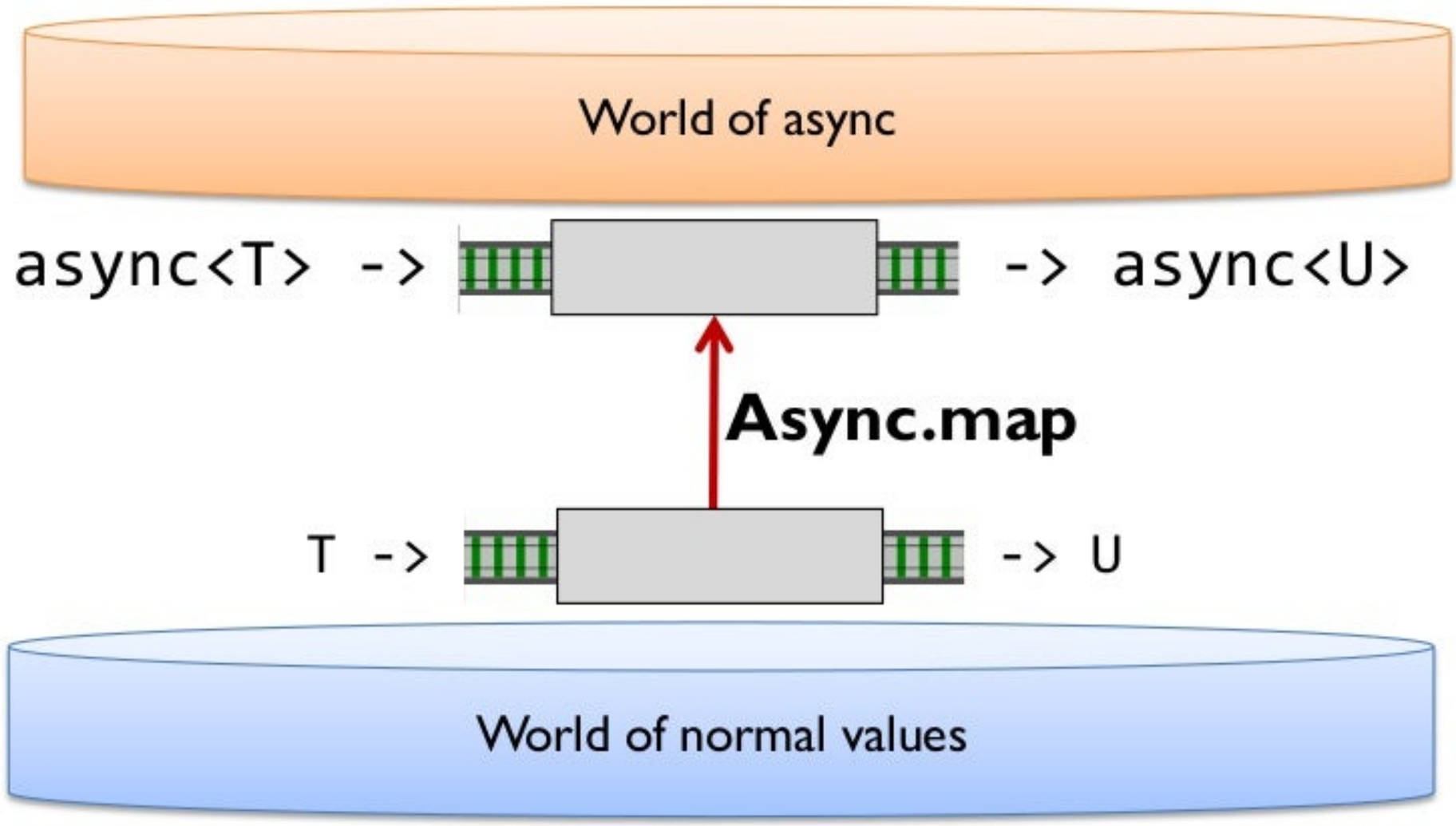
```fsharp
let add42 x = ...
```

A function in List world

```fsharp
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

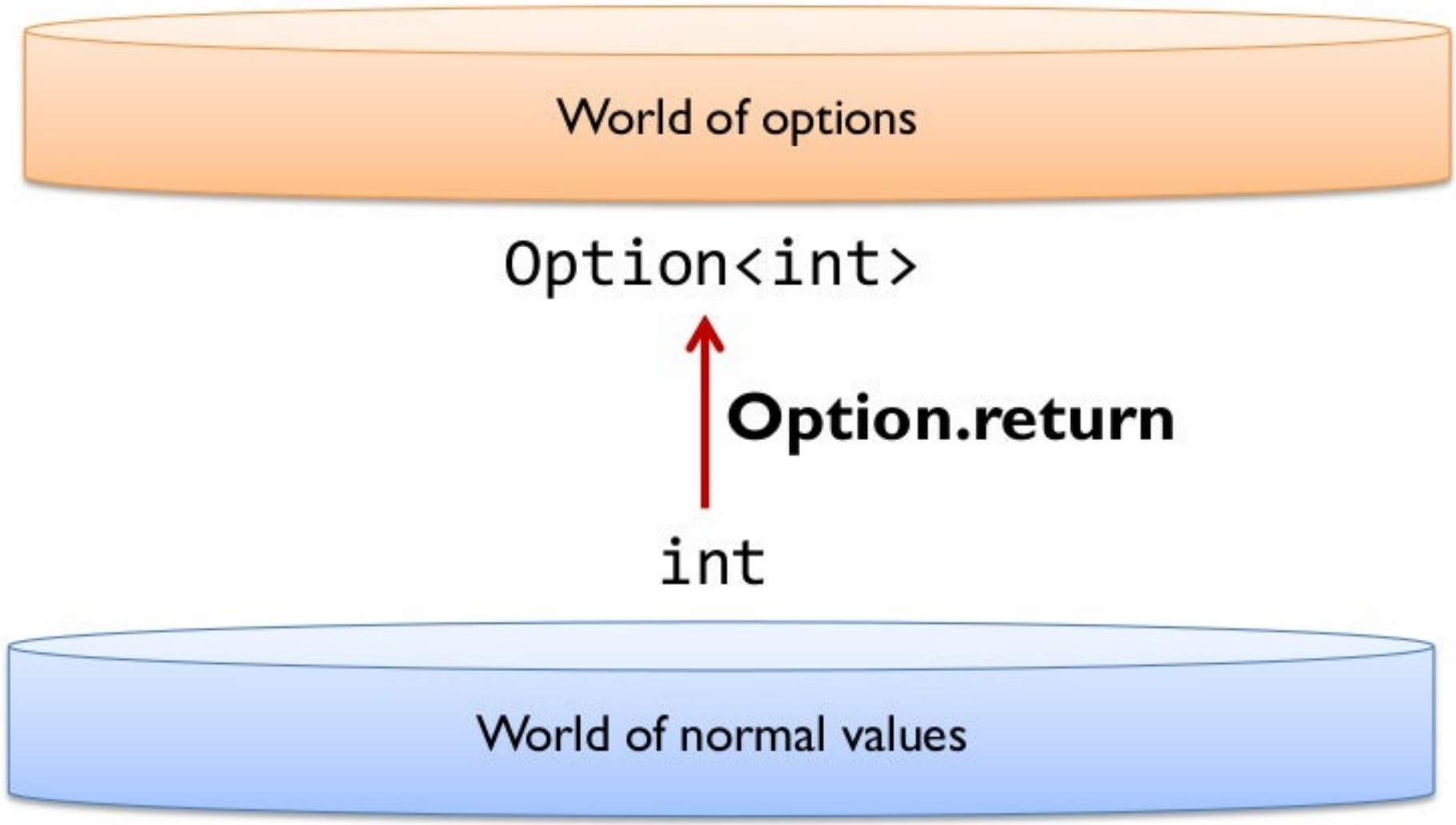
i. An effect type
   - e.g. `Option<>`, `List<>`, `Async<>`

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"

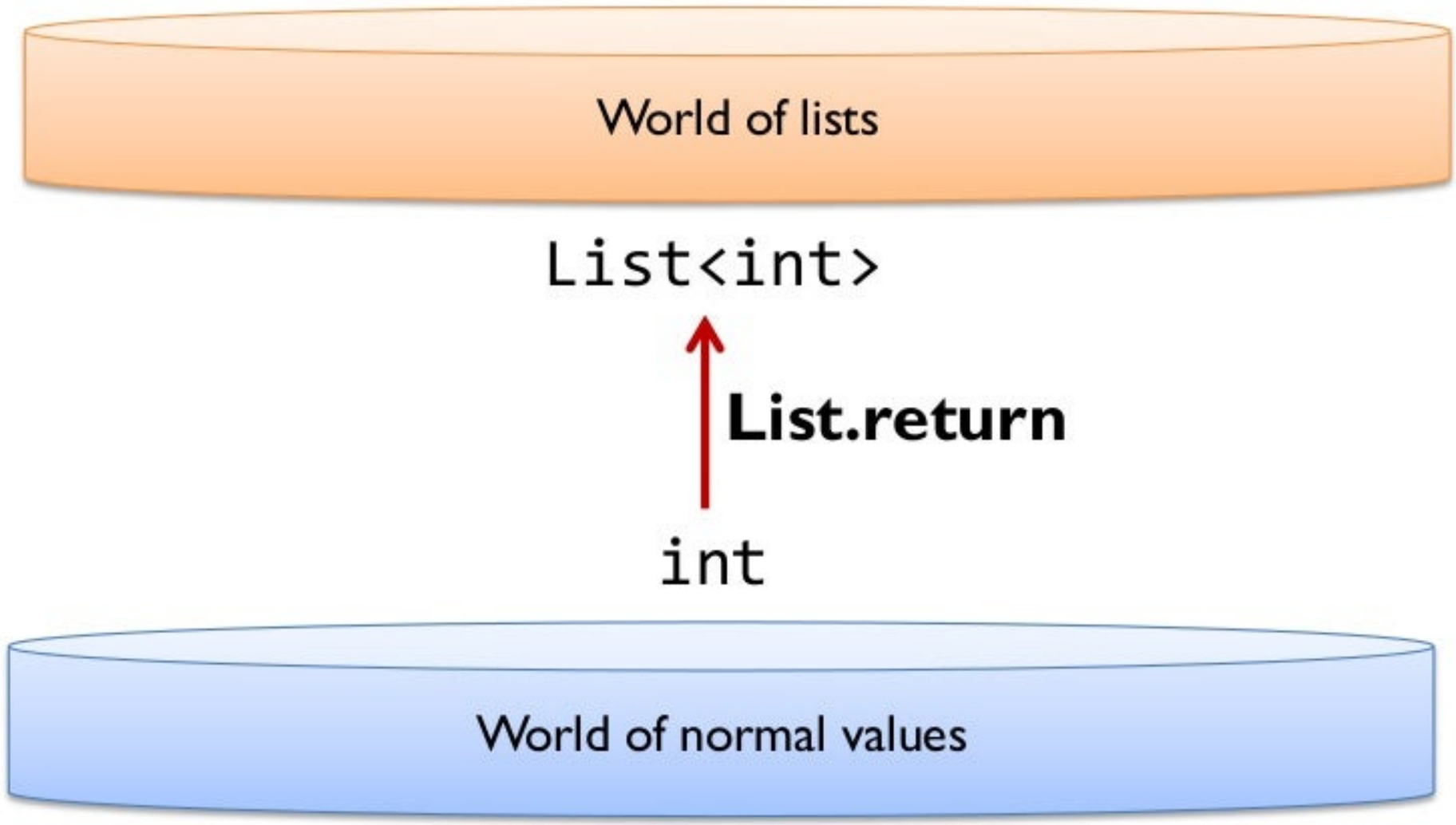

World of options
Option<int>
Option.return
int
World of normal values

A value in normal world

```
let x = 42
```

A value in Option world

```
let intOption = Some x
```


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

A value in normal world

```
let x = 42
```

A value in List world

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

Tool #4

Chaining world-crossing functions with "bind"

What's a world-crossing function?

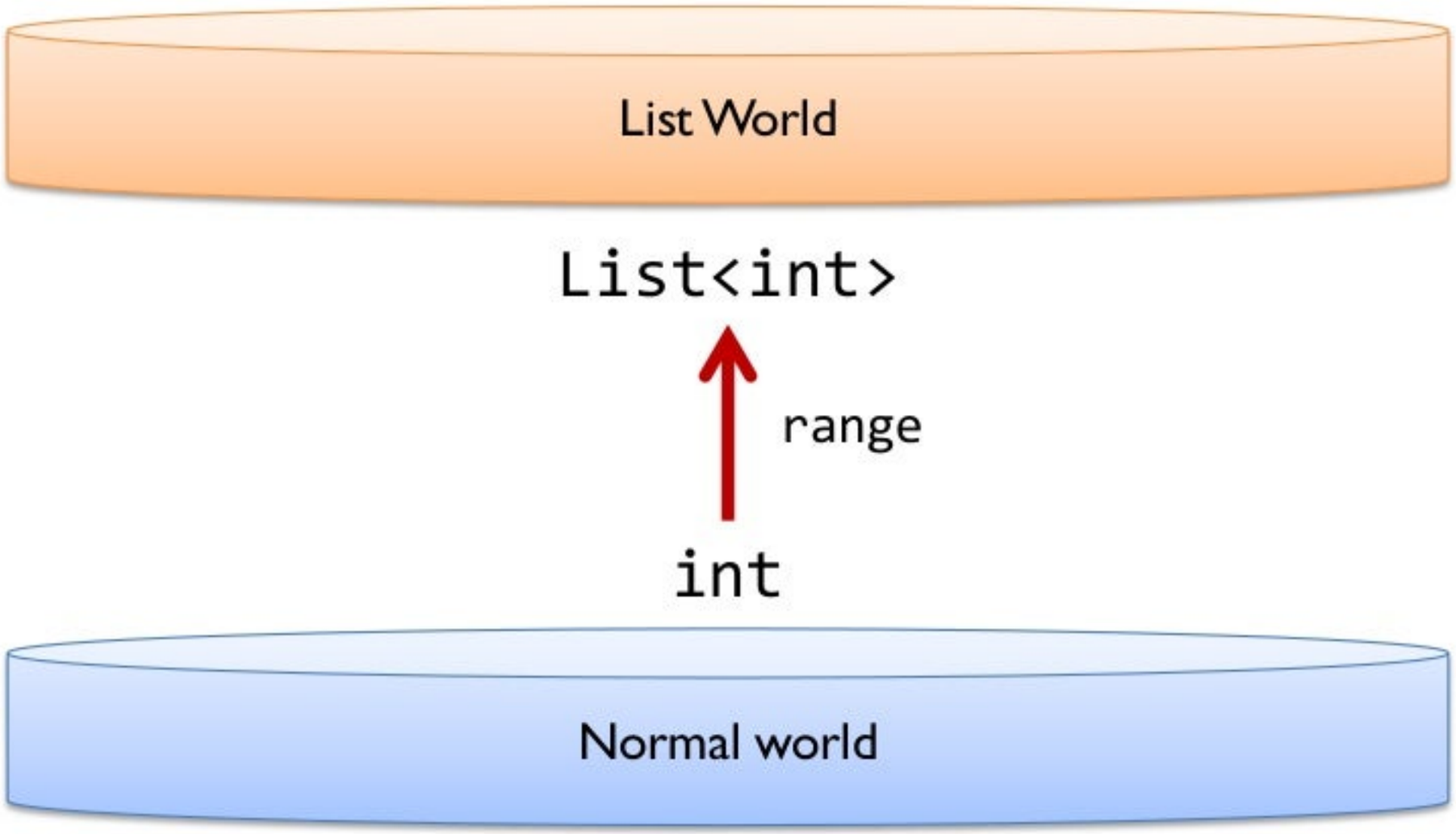
A value in normal world

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

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

A value in List world

A world crossing function



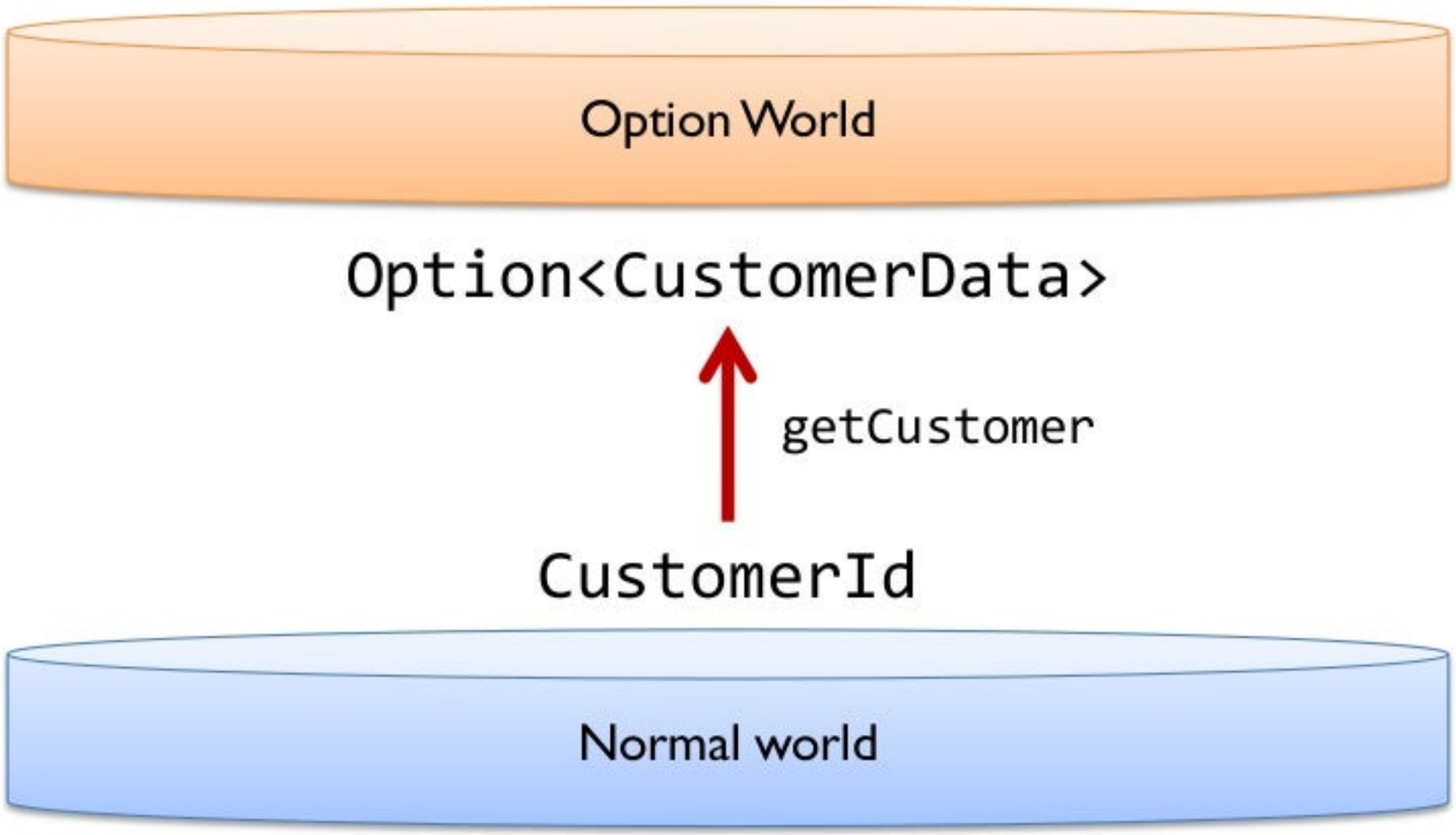
A value in normal world

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

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

A value in Option world

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
```

Parameterize all the things!

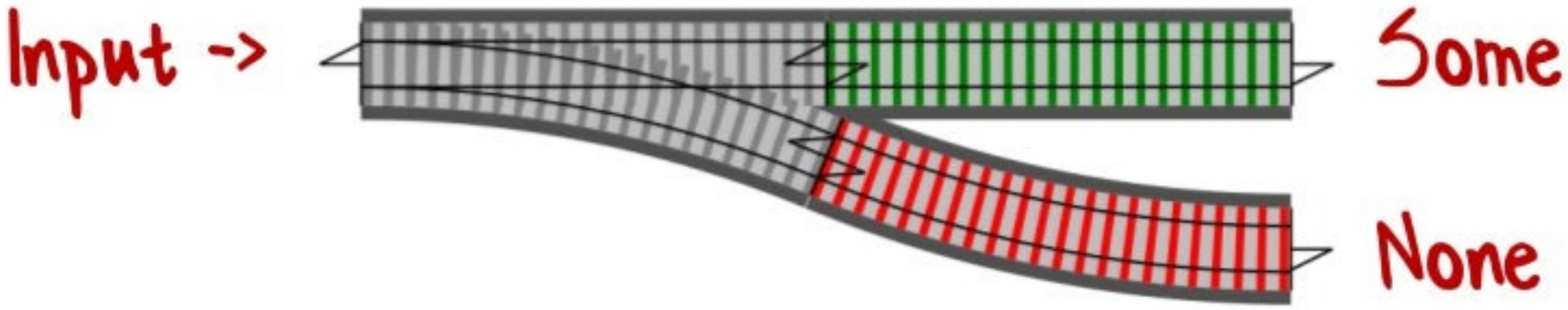
```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```

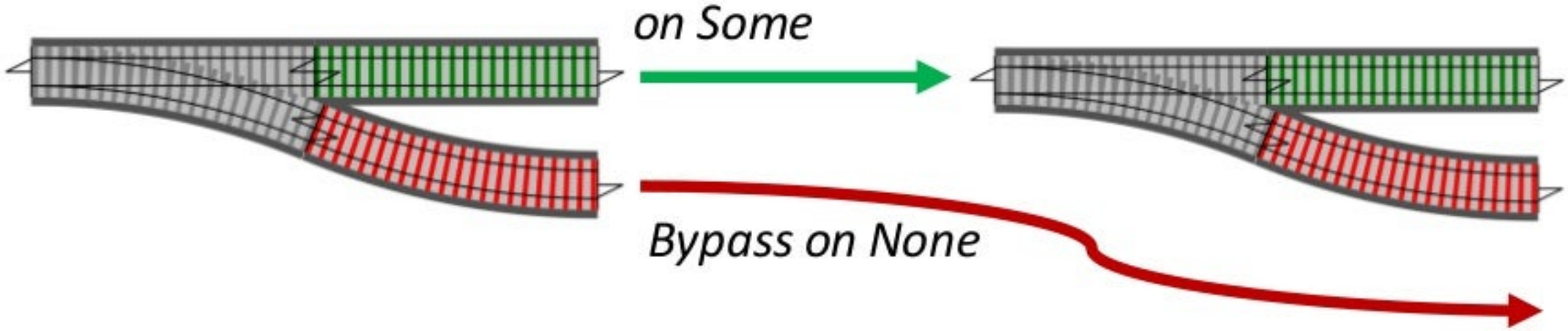
```fsharp
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```

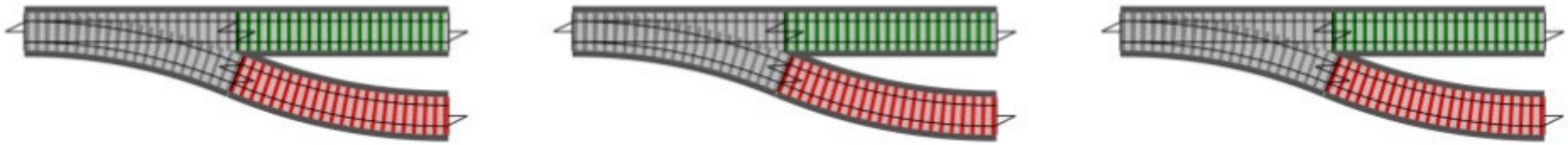
```fsharp
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo ...
```

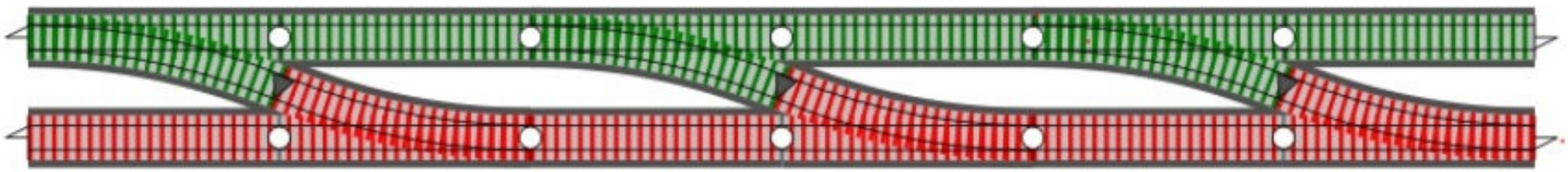
Much cleaner code now

Let's use a railway analogy

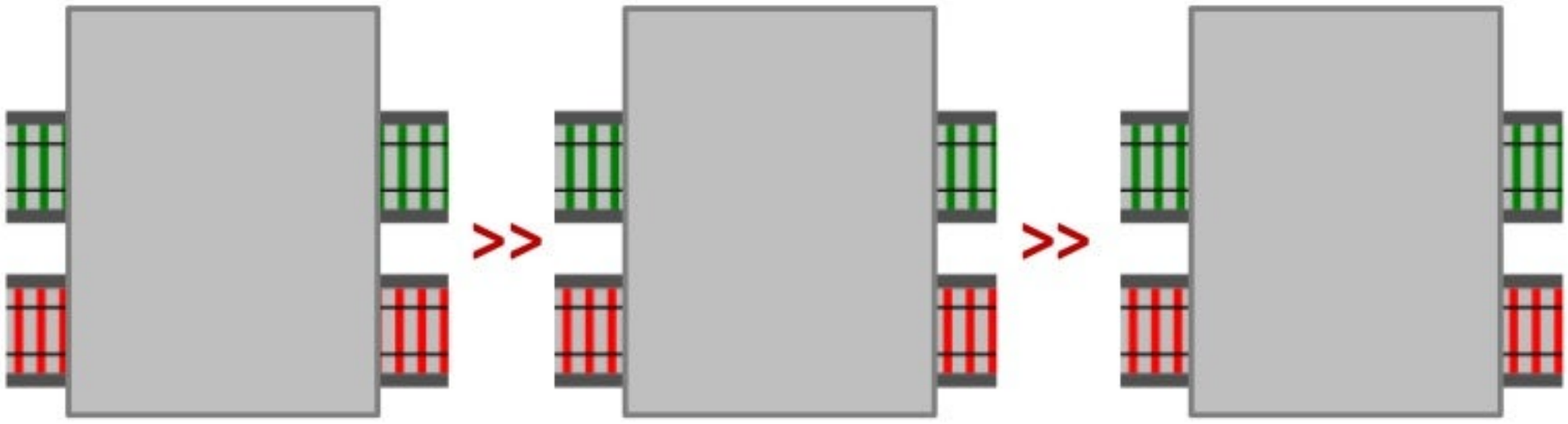



on Some
Bypass on None

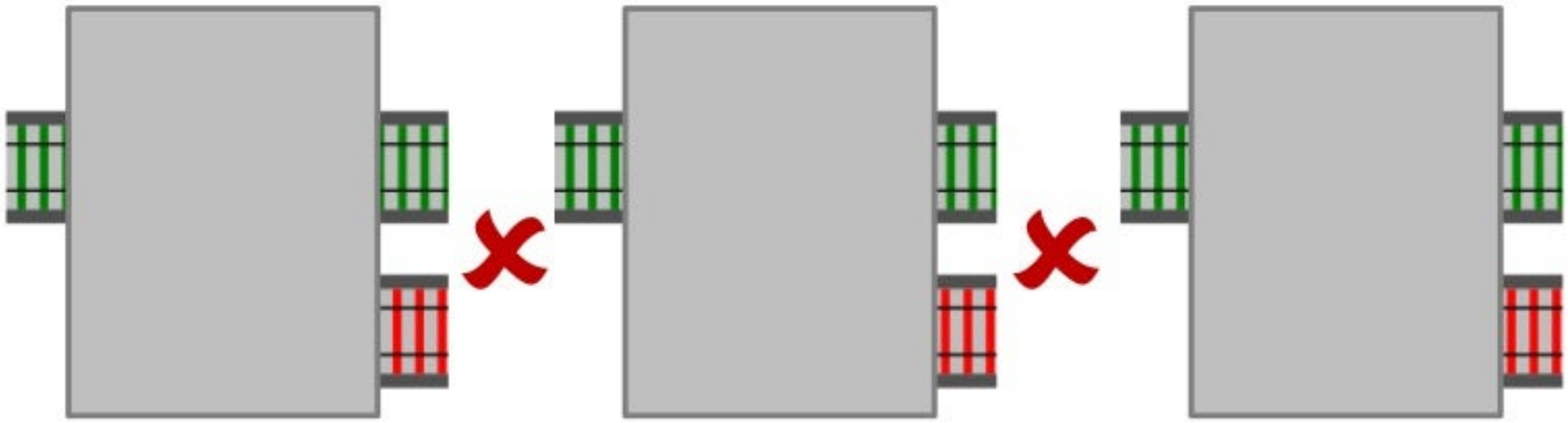




Composing one-track functions is fine...



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



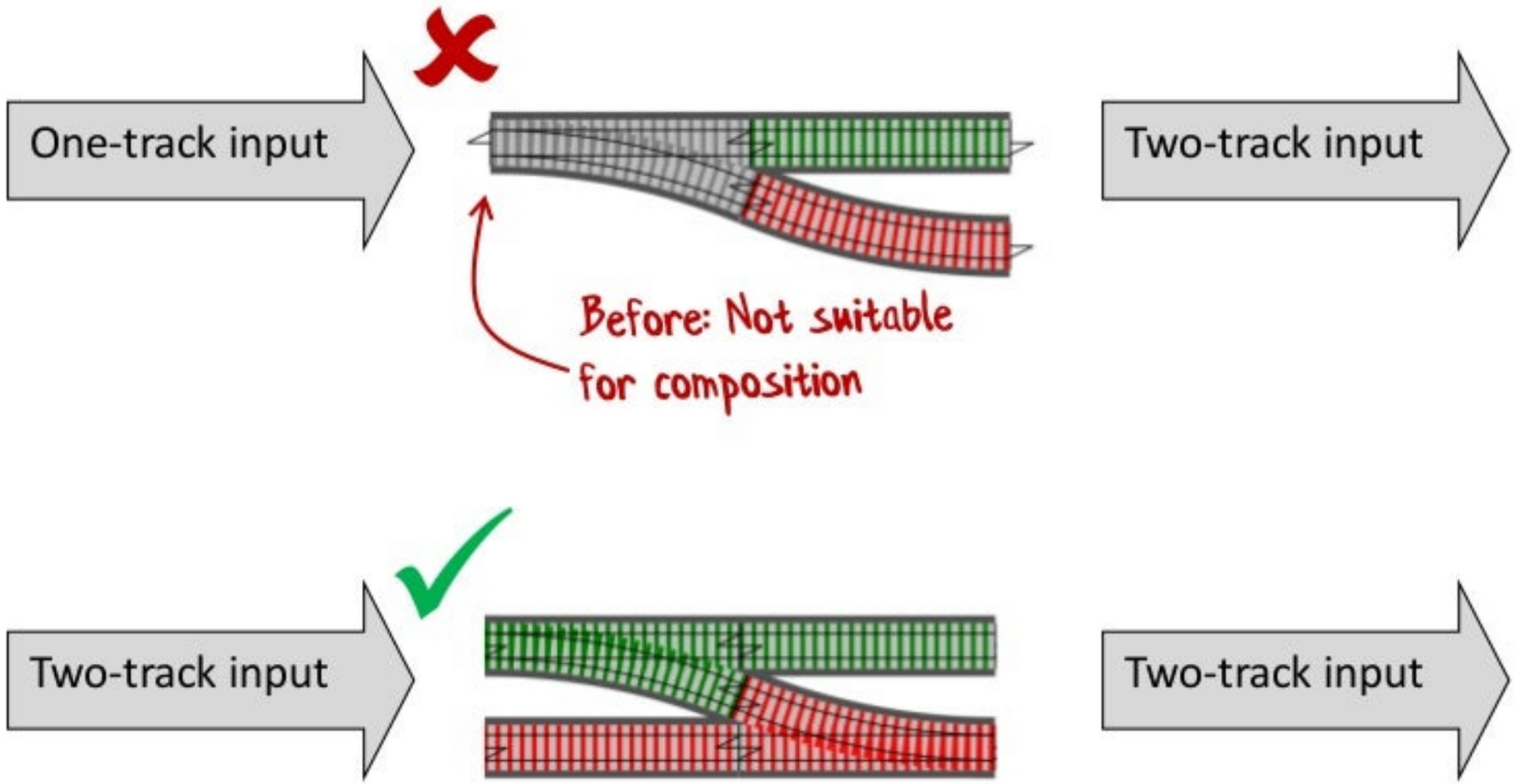
... but composing switches is not allowed!

How to combine the mismatched functions?

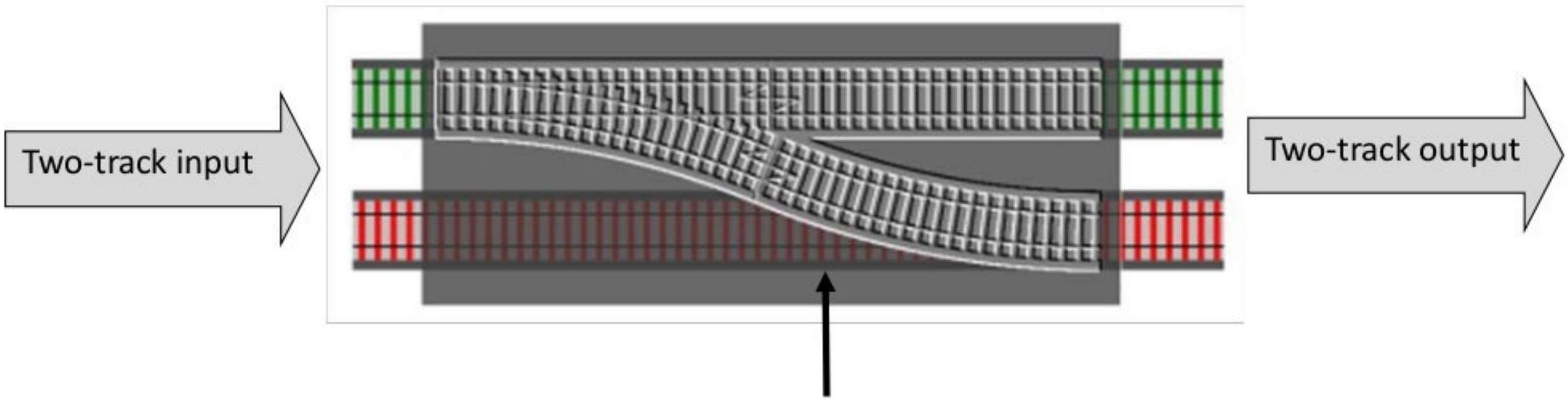
"Bind" is the answer!

Bind all the things!

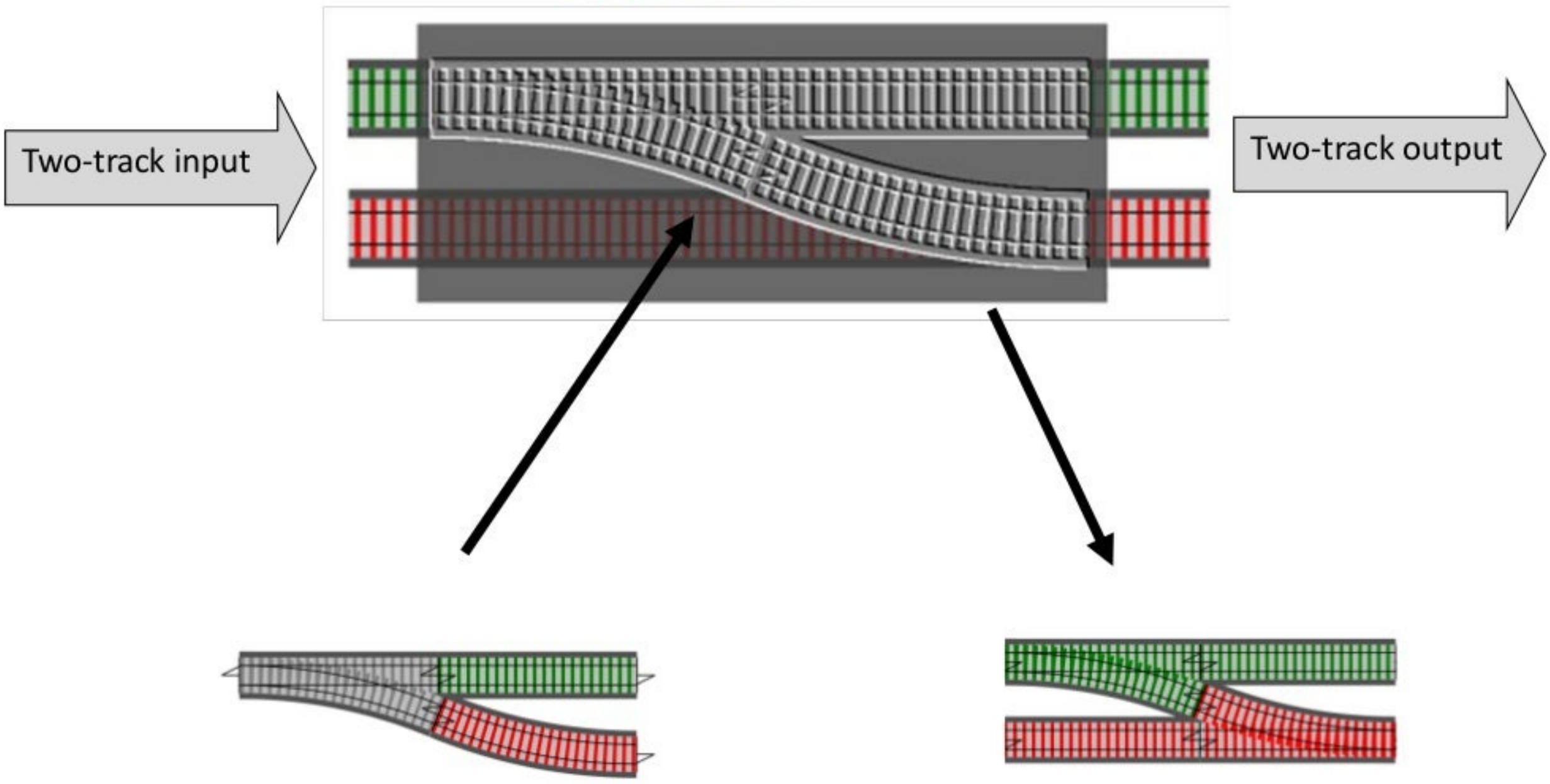
FP'ers get excited by bind

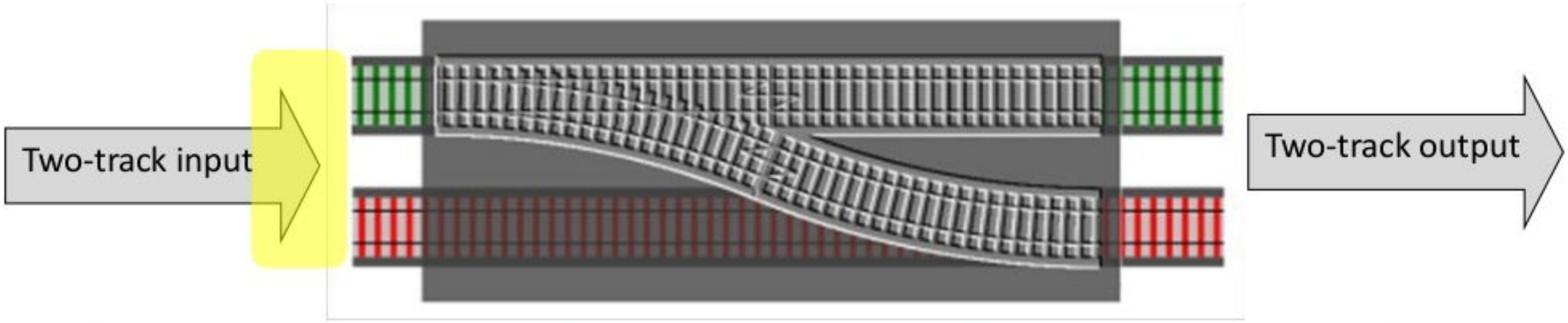


So how can we convert from the "before" case to the "after" case?

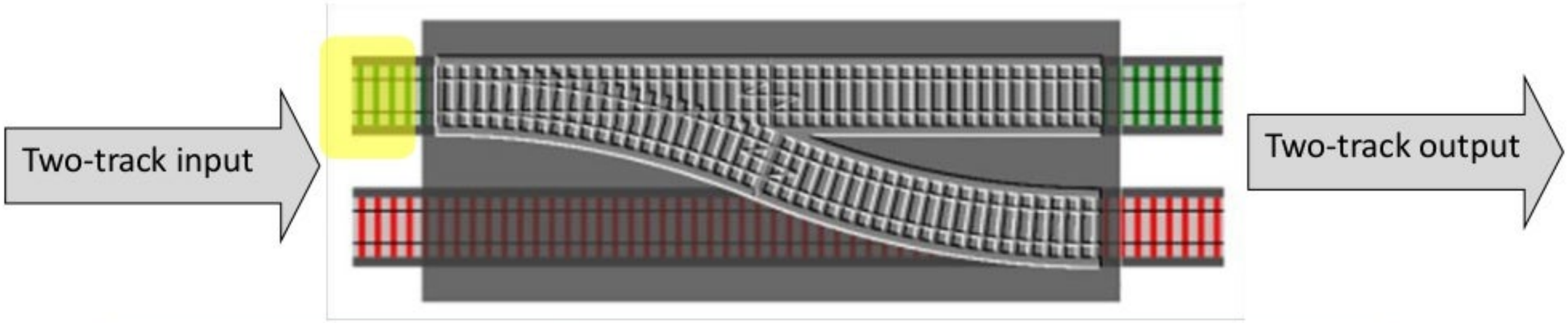

Two-track input
Two-track output
Slot for switch function

A "function transformer"

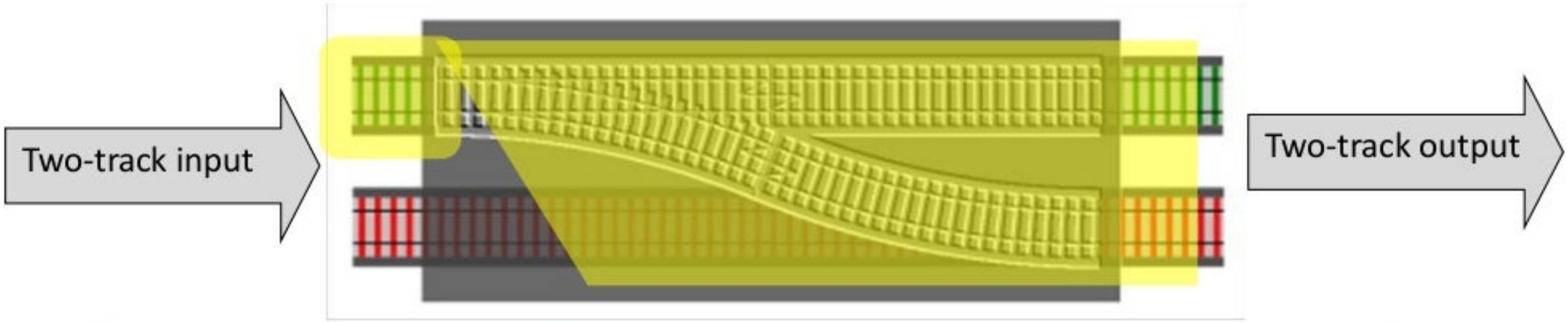




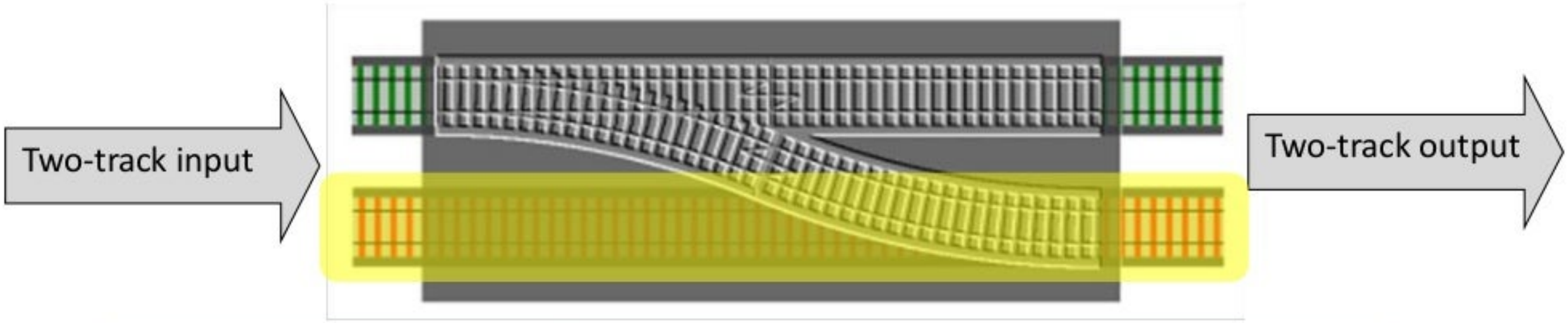

```
let bind nextFunction optionInput =
    match optionInput with
    | Some s -> nextFunction s
    | None -> None
```




```
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
```

After

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

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

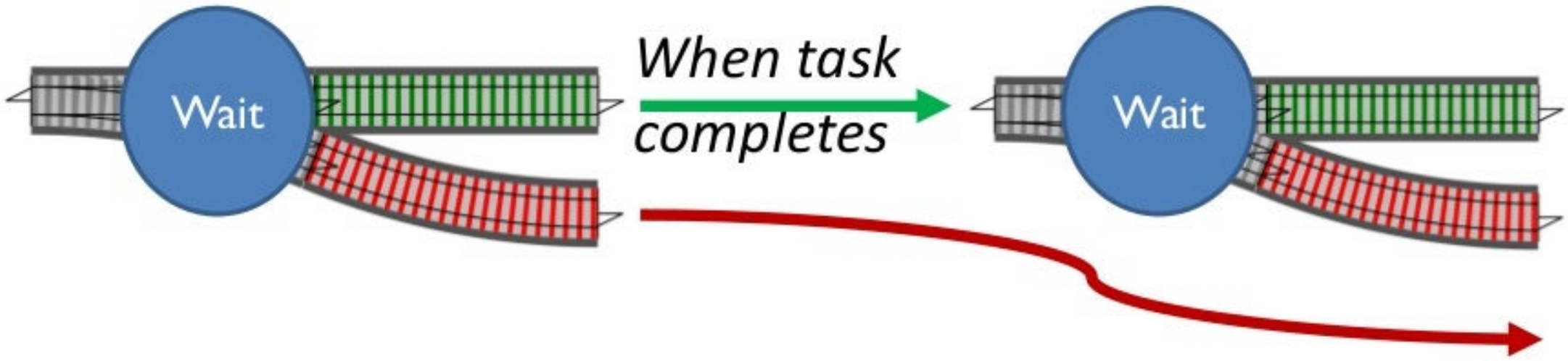
No pyramids!

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?

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  validateRequest(request);
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  var result = db.updateDbFromRequest(request);
  if (!result) {
    return "Customer record not found"
  }

  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  try {
    var result = db.updateDbFromRequest(request);
    if (!result) {
      return "Customer record not found"
    }
  } catch {
    return "DB error: Customer record not updated"
  }

  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  try {
    var result = db.updateDbFromRequest(request);
    if (!result) {
      return "Customer record not found"
    }
  } catch {
    return "DB error: Customer record not updated"
  }

  if (!smtpServer.sendEmail(request.Email)) {
    log.Error "Customer email not sent"
  }

  return "OK";
}
```

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  try {
    var result = db.updateDbFromRequest(request);
    if (!result) {
      return "Customer record not found"
    }
  } catch {
    return "DB error: Customer record not updated"
  }

  if (!smtpServer.sendEmail(request.Email)) {
    log.Error "Customer email not sent"
  }

  return "OK";
}
```

6 clean lines -> 18 ugly lines. 200% extra!
Sadly this is typical of error handling code.

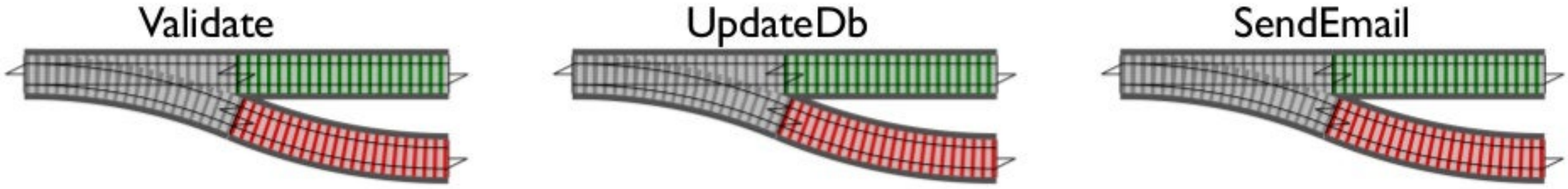
Solution:

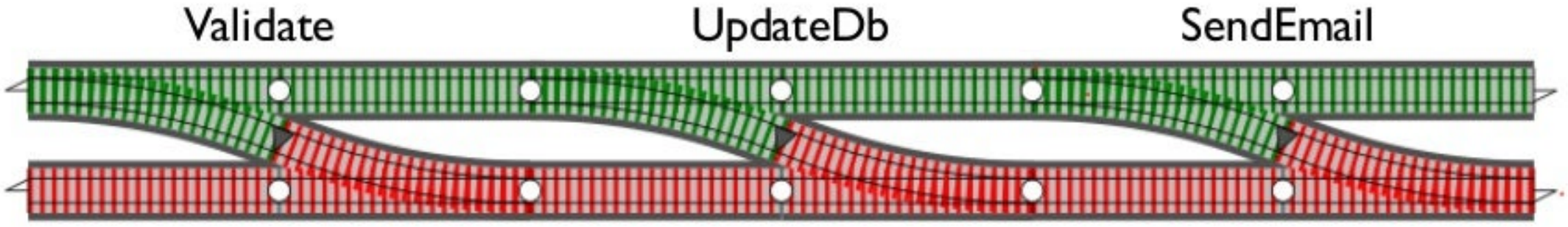
Use a *Result* type for error handling


```
type Result =
    | Ok of SuccessValue
    | Error of ErrorValue
```

Define a choice type


```
let validateInput input =
    if input.name = "" then
        Error "Name must not be blank"
    else if input.email = "" then
        Error "Email must not be blank"
    else
        Ok input  // happy path
```


Validate
UpdateDb
SendEmail



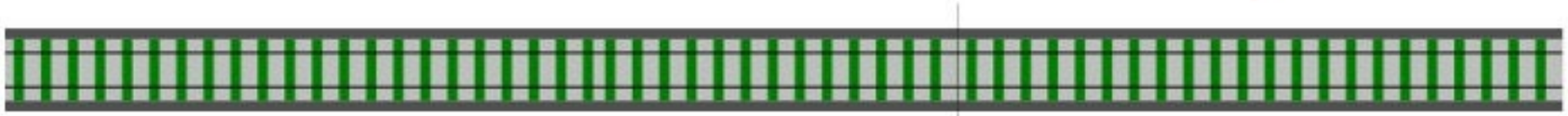
This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

Before

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

Functional flow without error handling

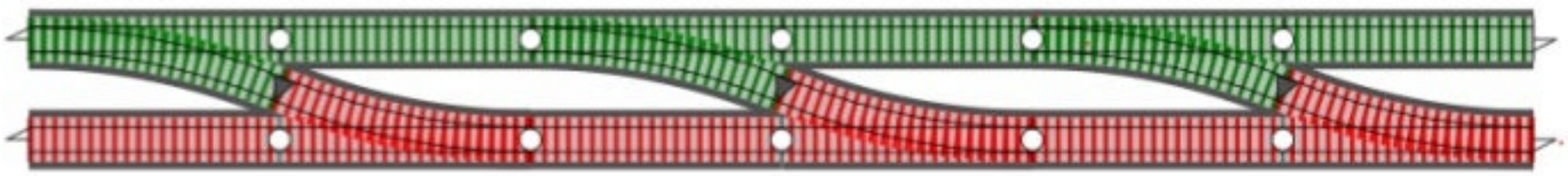
One track



After

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

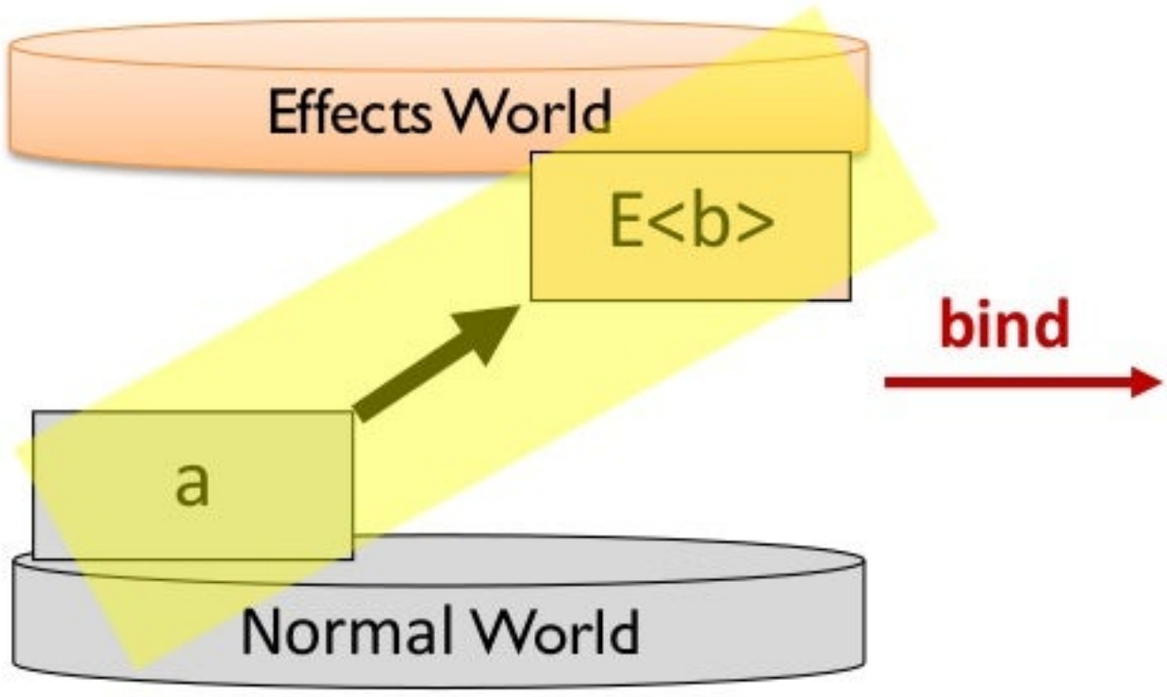
Two track



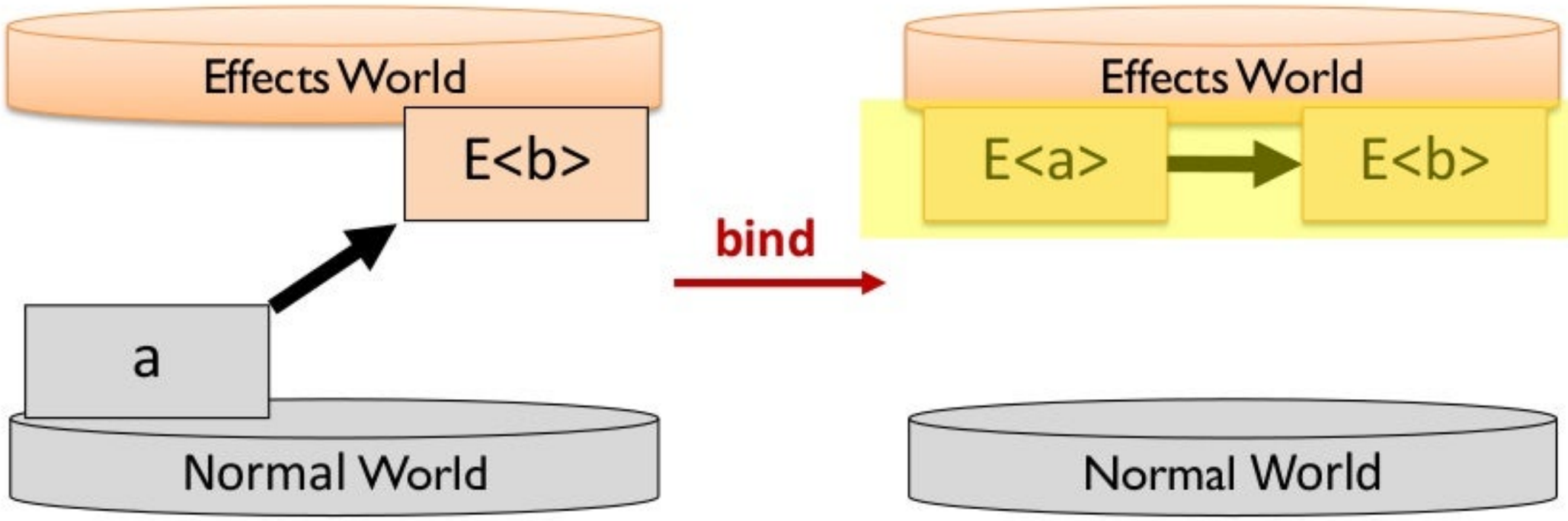
See fsharpforfunandprofit.com/rop

Why is bind so important?

It makes world-crossing functions composable



Before bind:
A diagonal function
(world crossing)

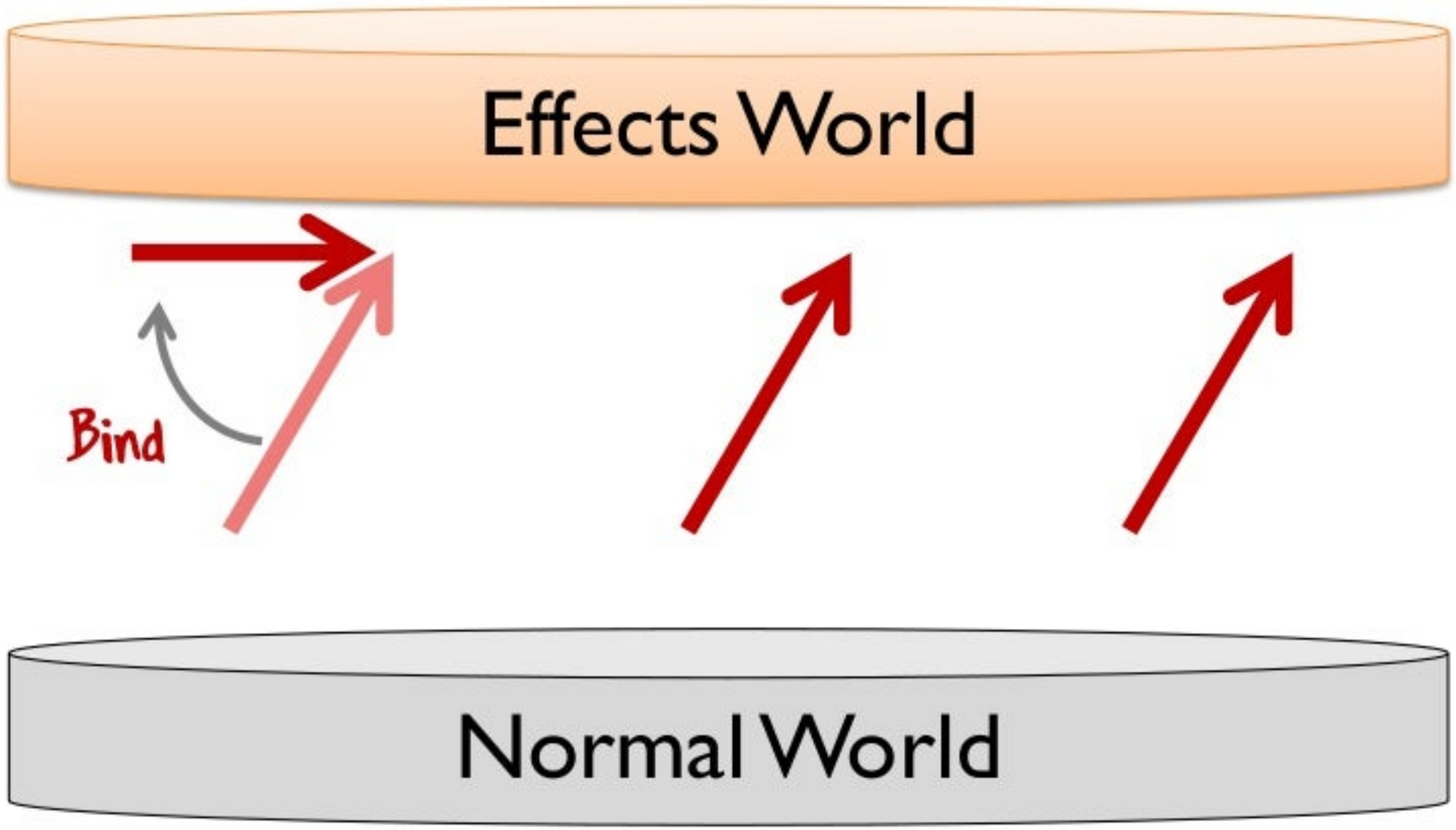


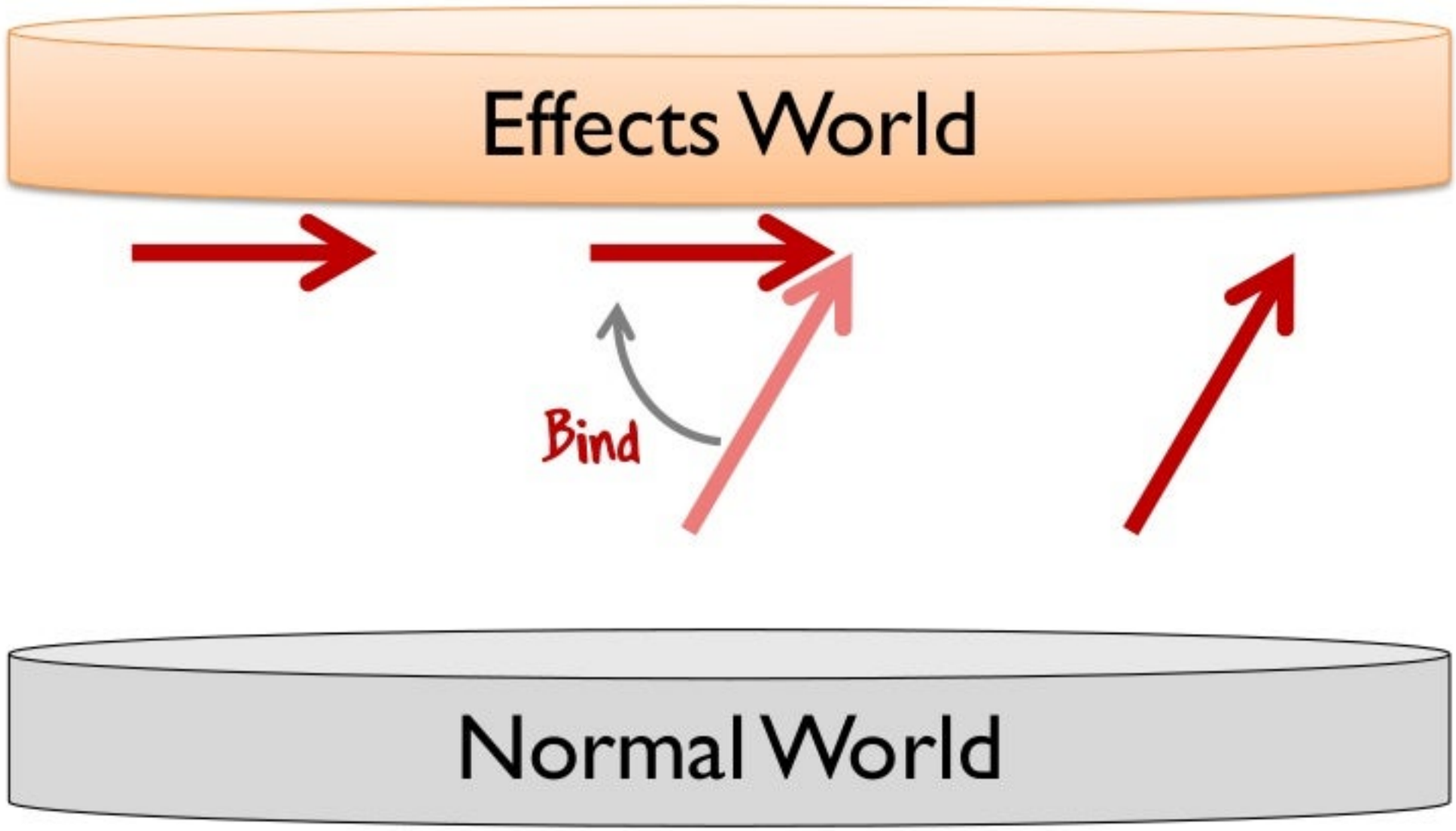
After bind:
A horizontal function
(all in E-world)

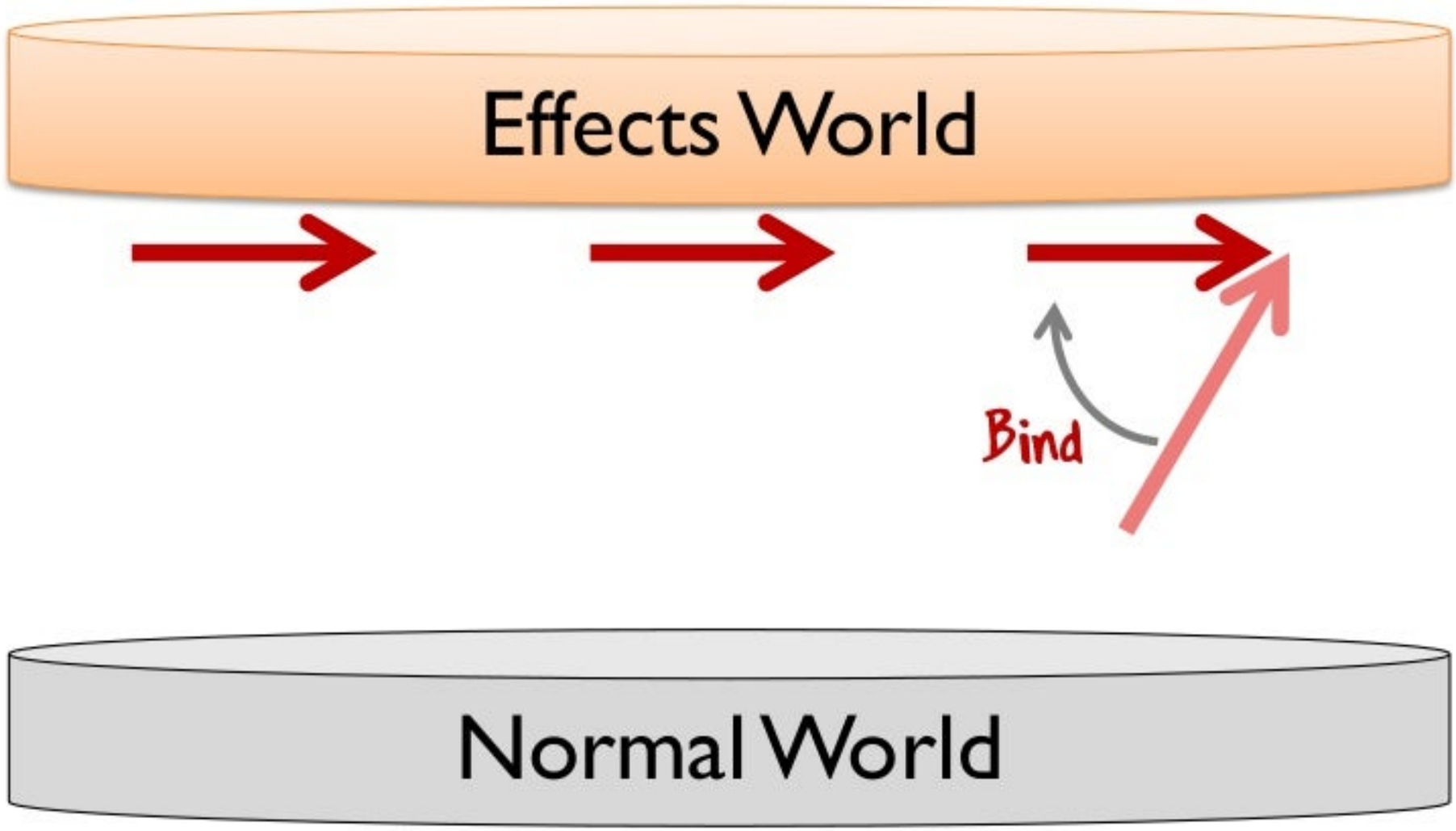
Effects World

"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

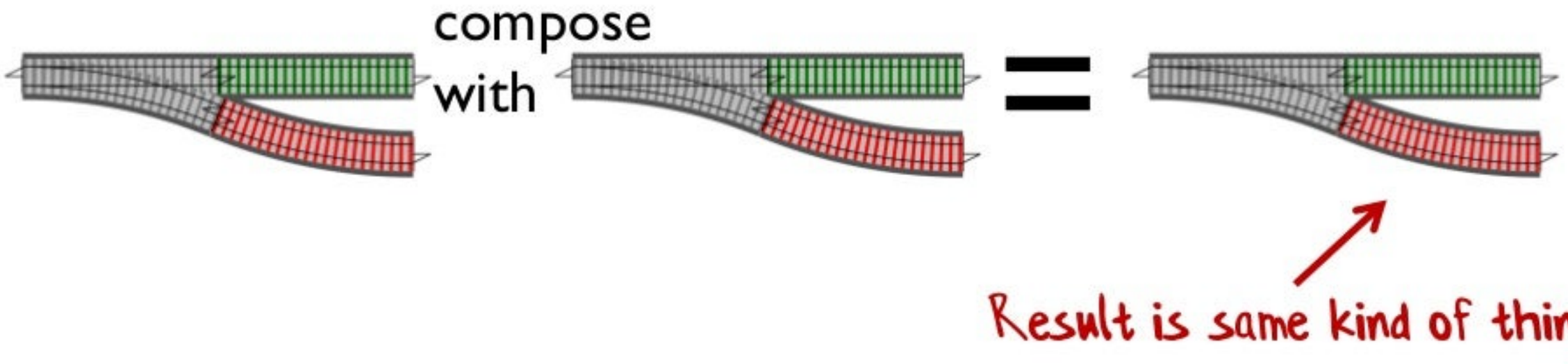
TLDR: If you want to chain effects-generating functions in series, use a **Monad**

Monads vs. monoids?

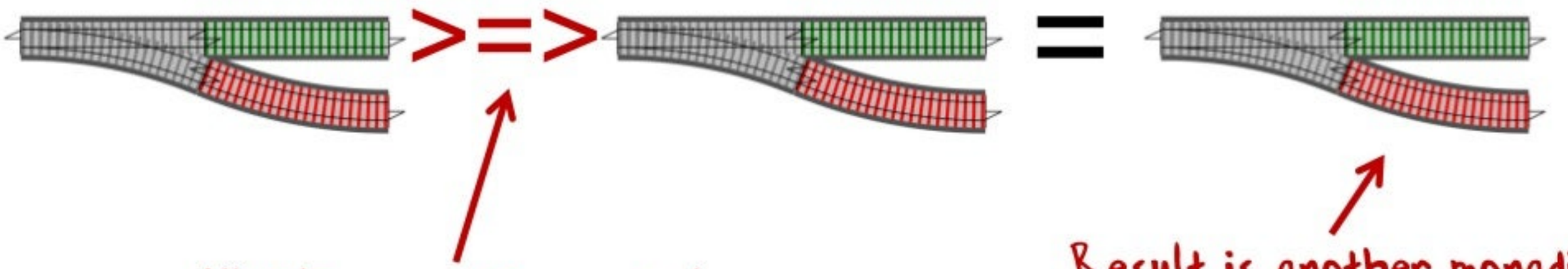
Confusing!

But there is a connection...

Kleisli Composition



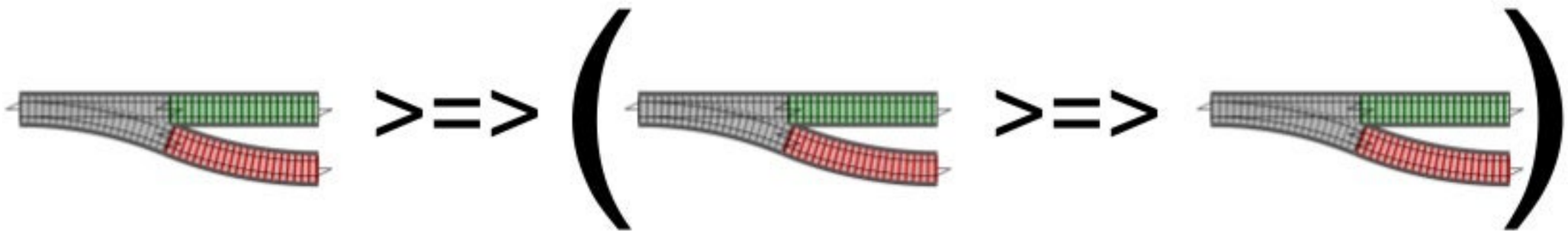
Kleisli Composition

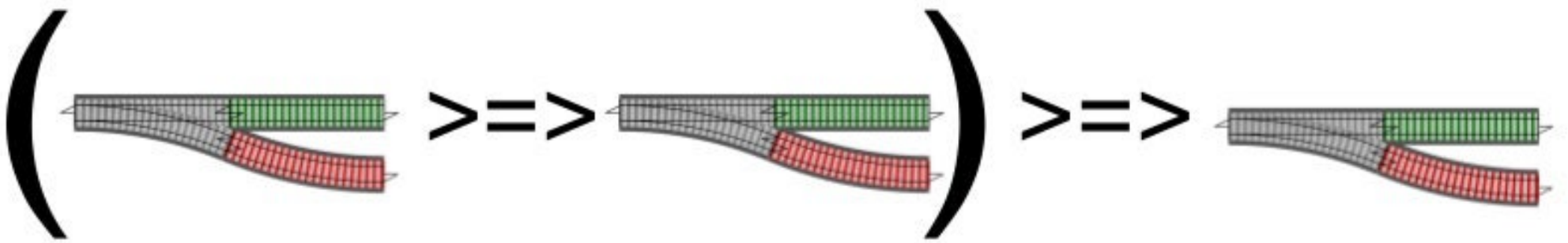


Kleisli composition symbol

Result is another monadic function so you can repeat.

Sound familiar?





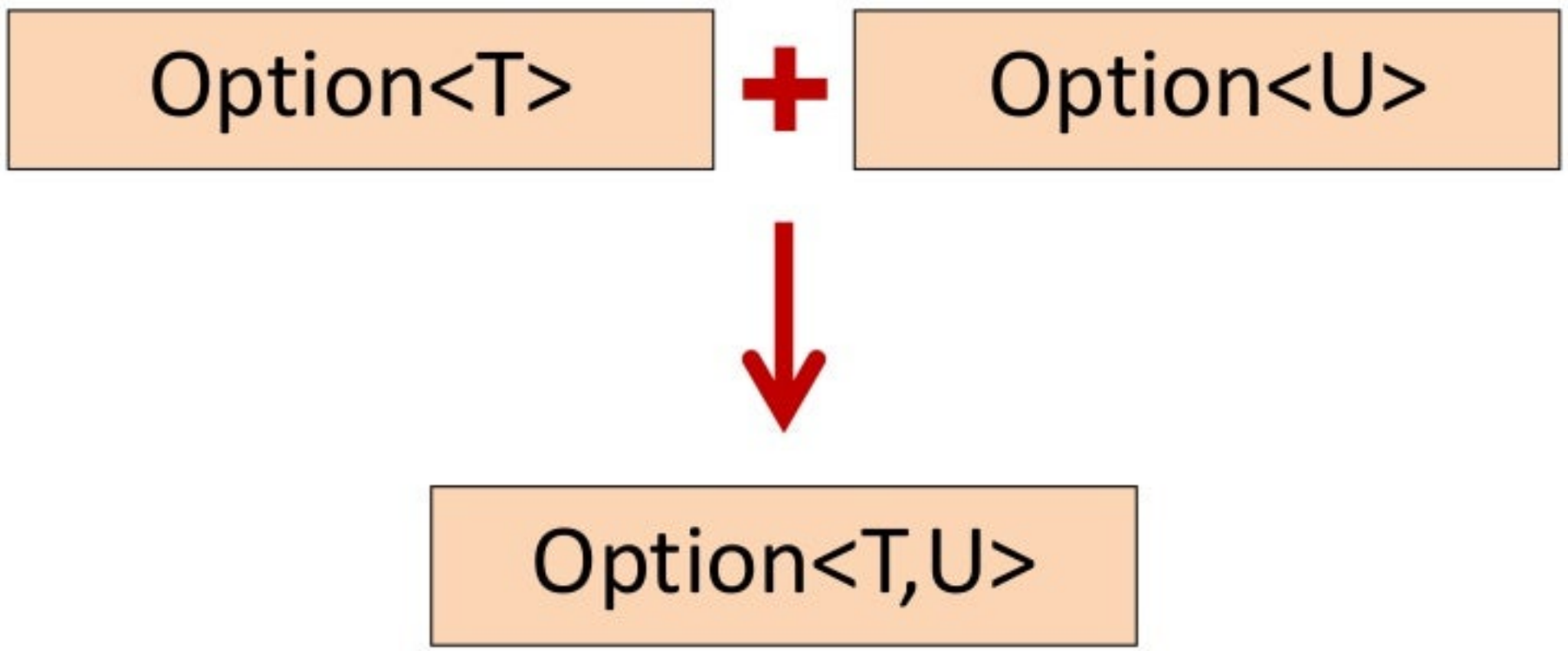
Order not important (Associative)

Monoid!

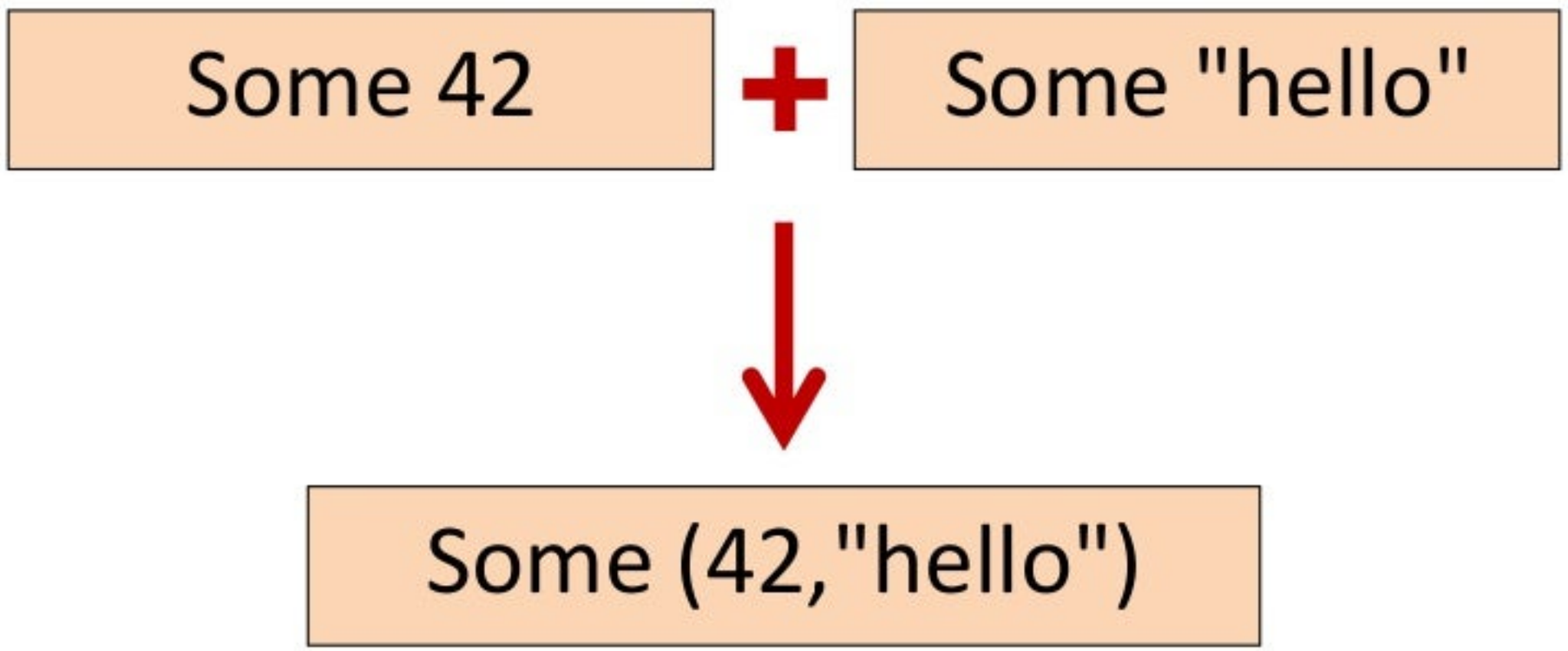
Tool #5

Combining effects in parallel with applicatives

How to combine effects?

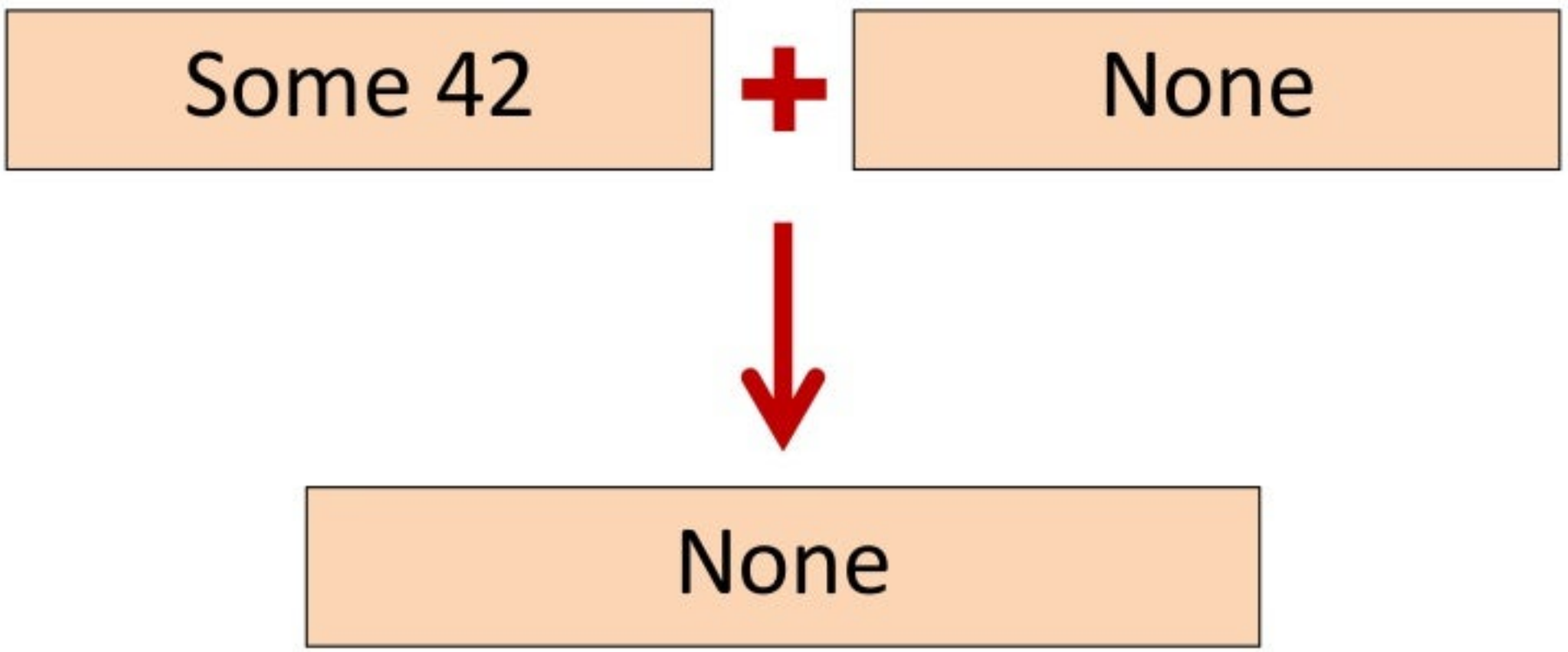


Combining options

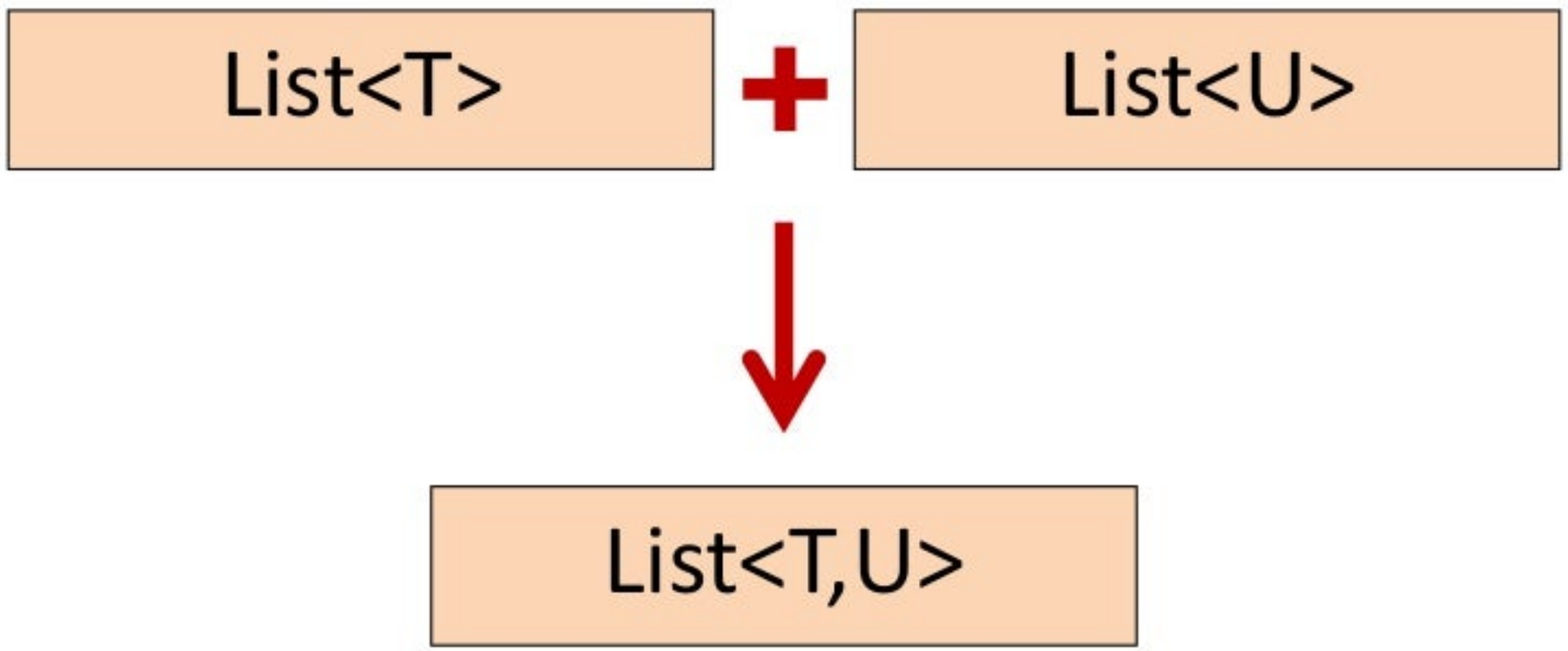


This is what you expect!

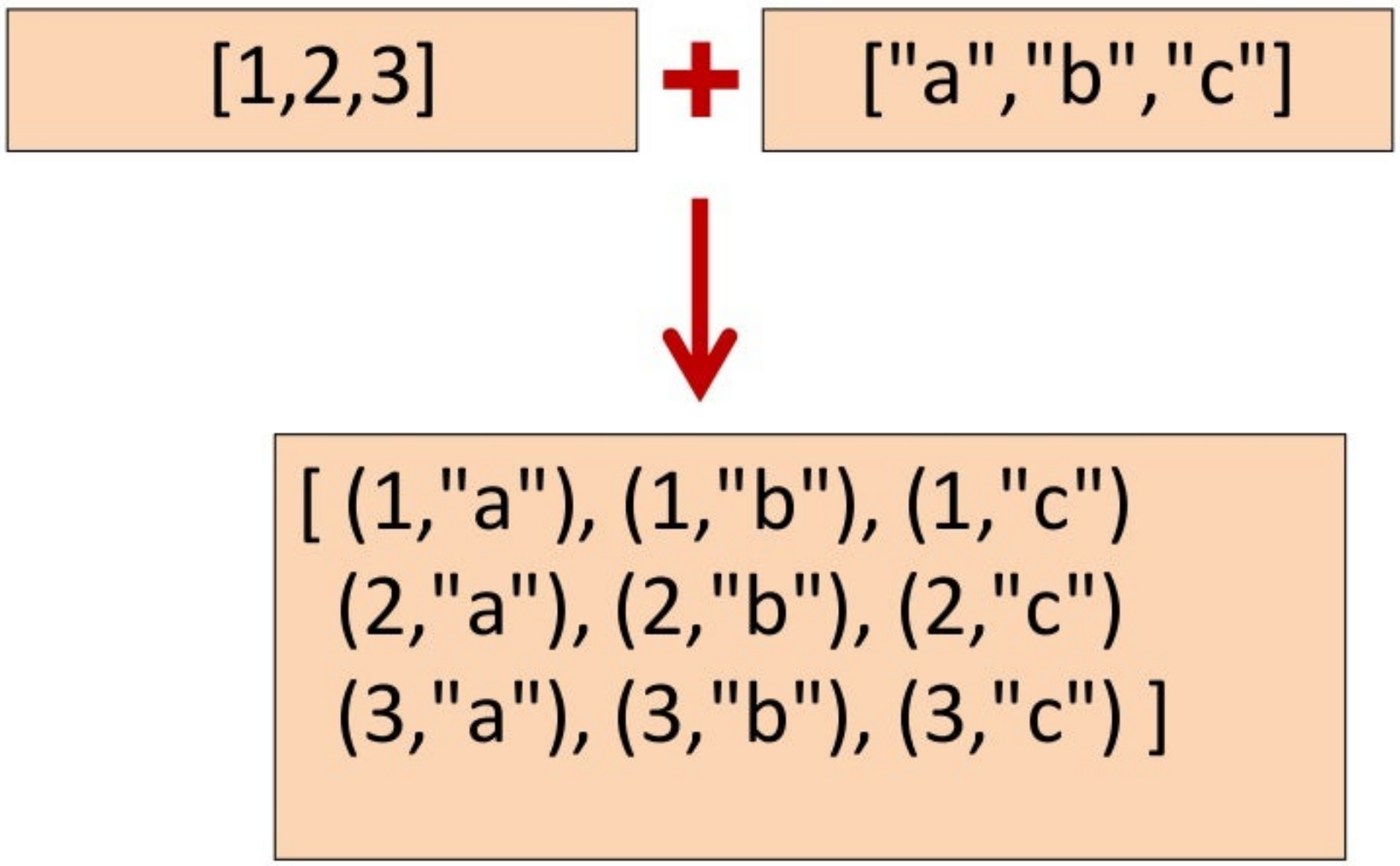
Combining options



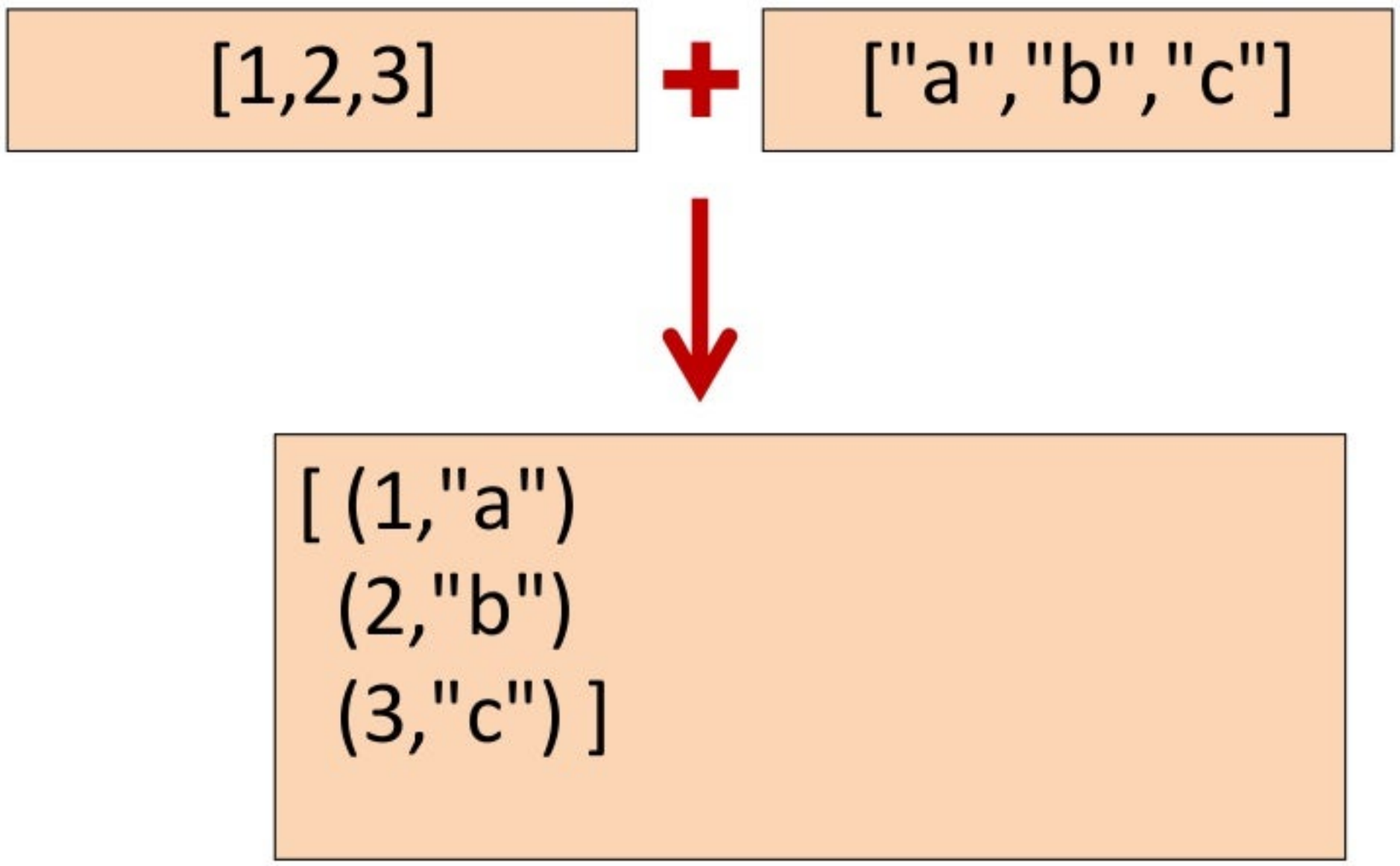
How to combine Lists?



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:

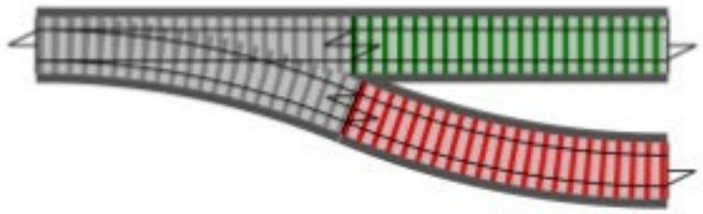
How to validate multiple fields in parallel?

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

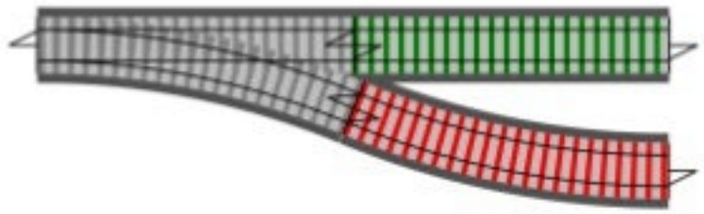
Each field must be validated

So we create some validation functions:

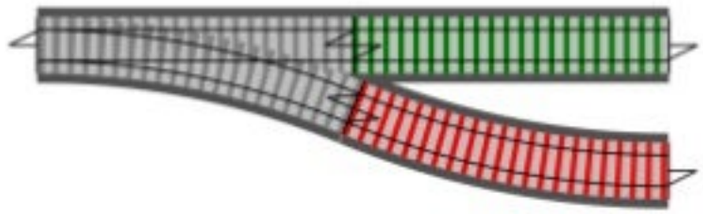
validateName



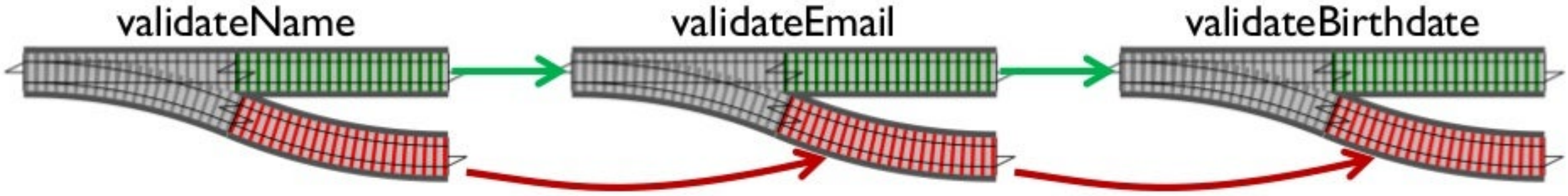
validateEmail



validateBirthdate

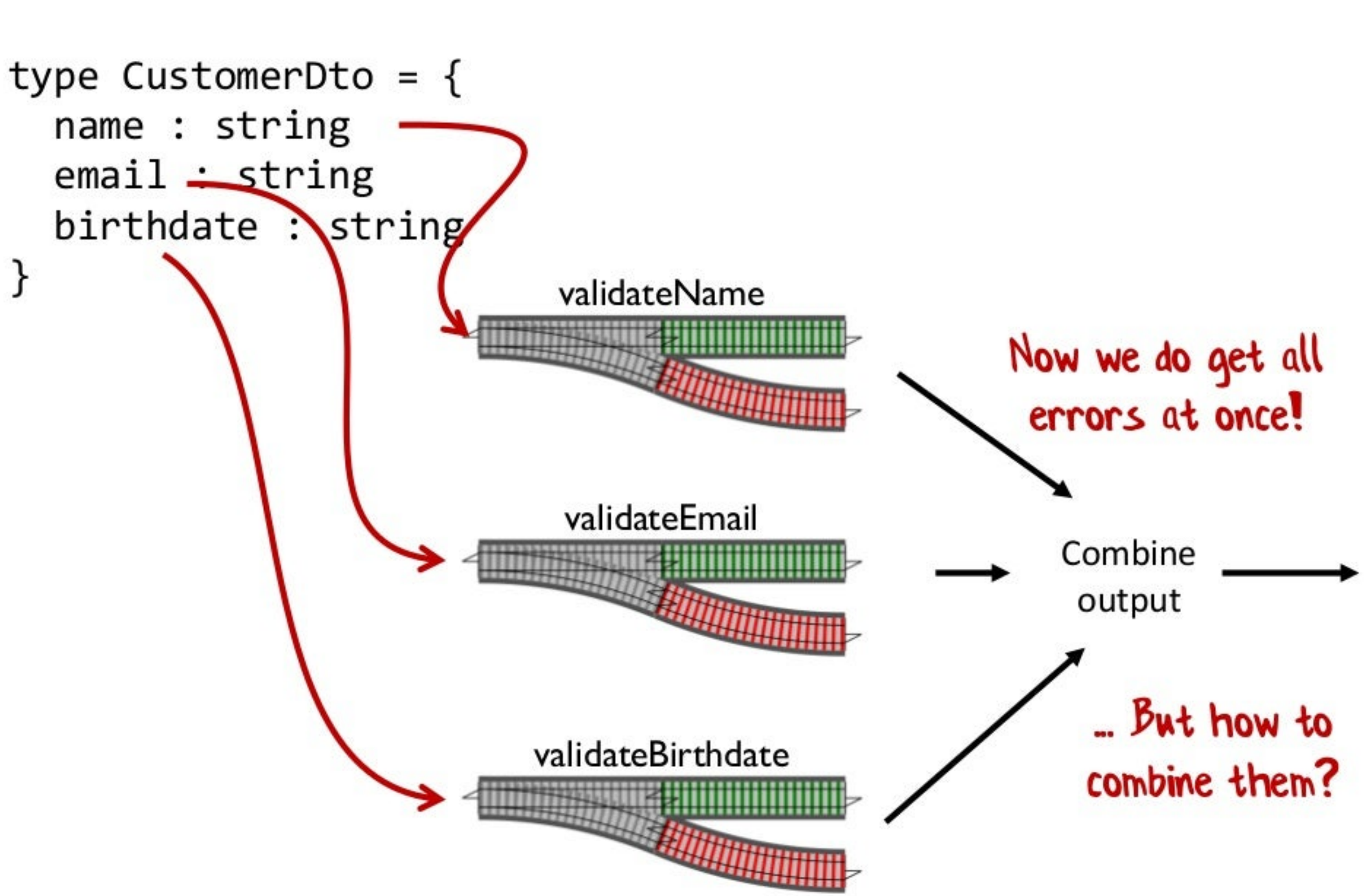


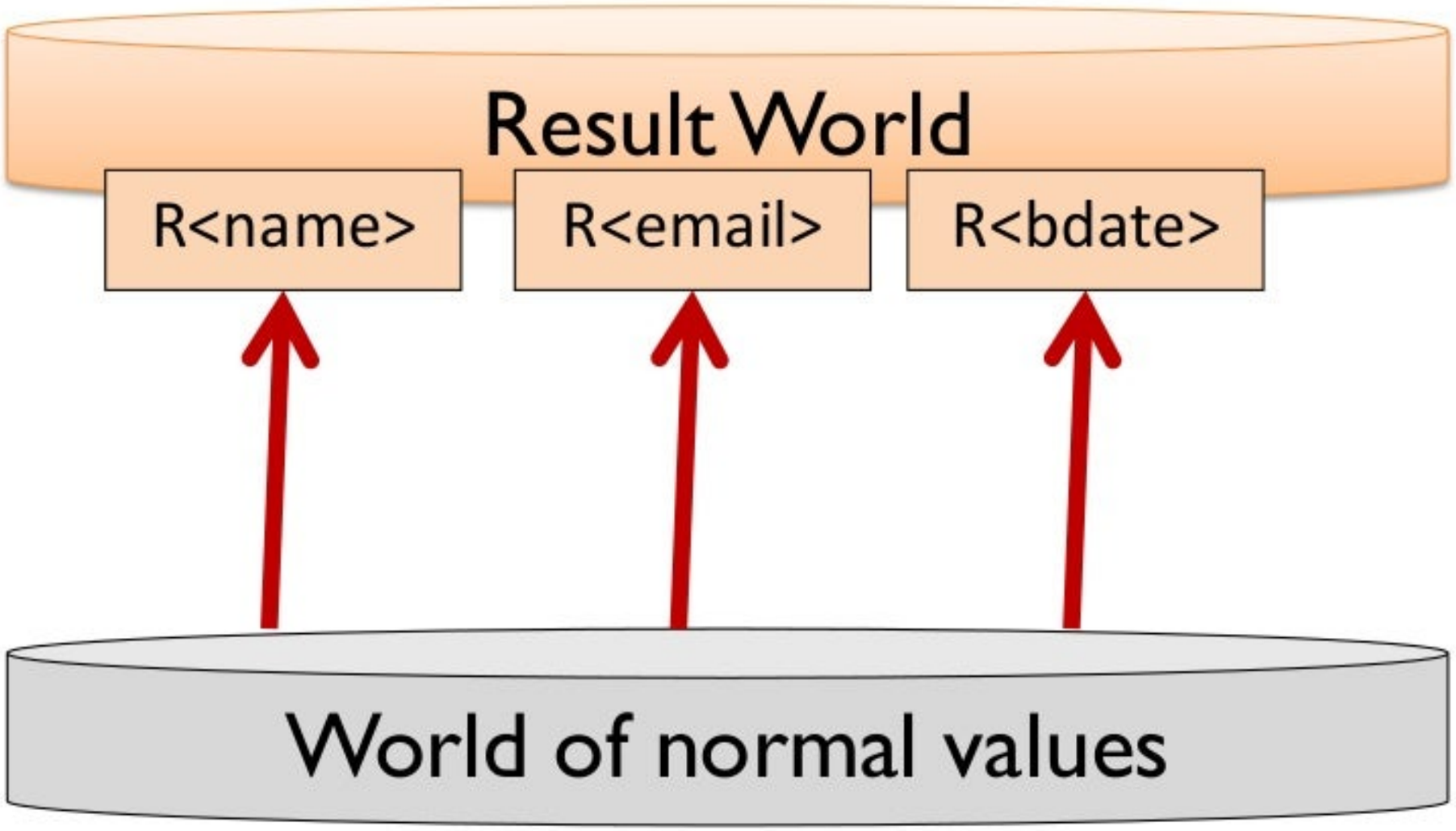
Problem: Validation done in series.
So only one error at a time is returned



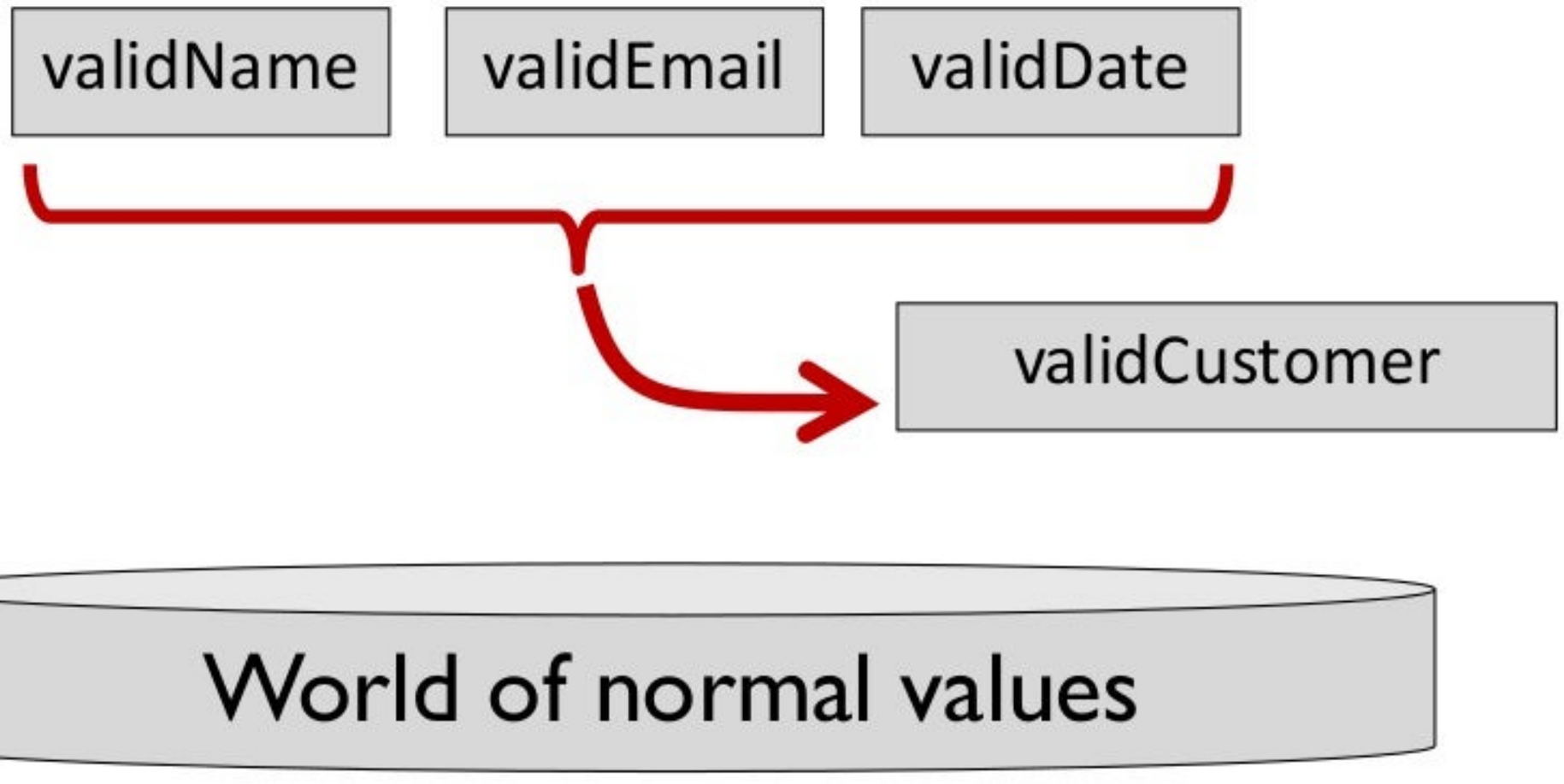
It would be nice to return all validation errors at once.

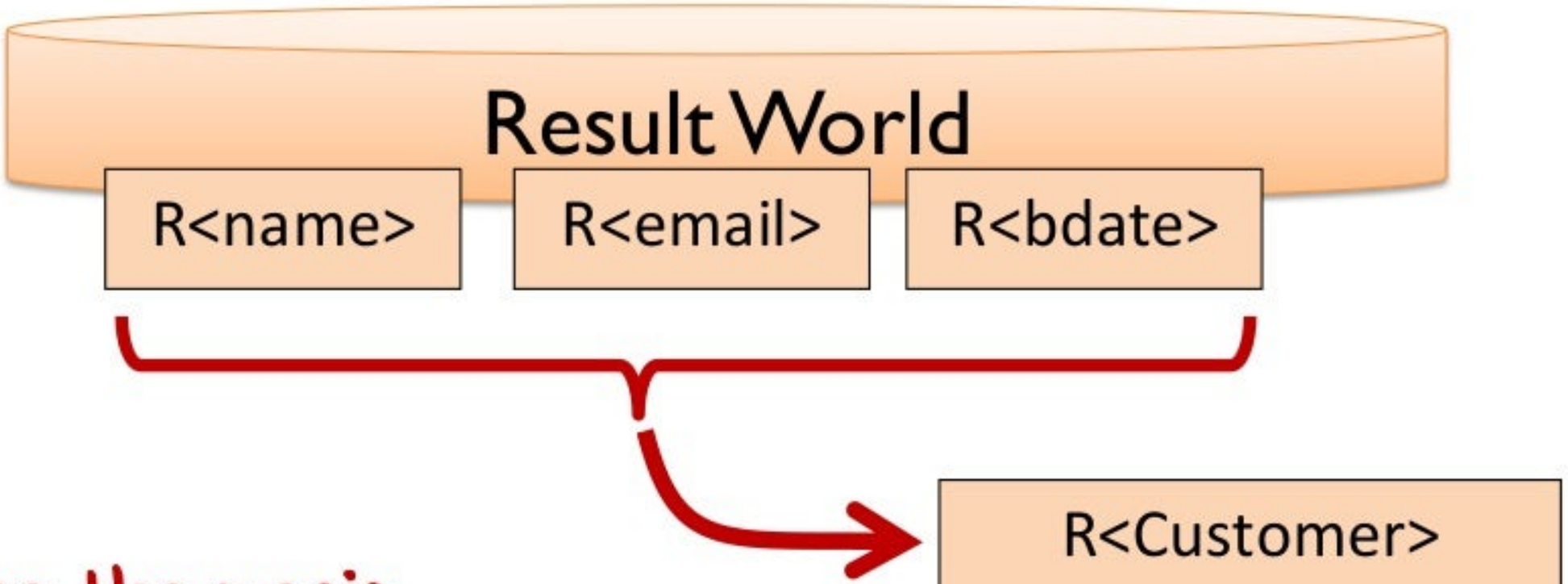
```
type CustomerDto = {
   name : string
   email : string
   birthdate : string
}
```




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

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




Result World
R<name>
R<email>
R<bdate>
R<Customer>
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“

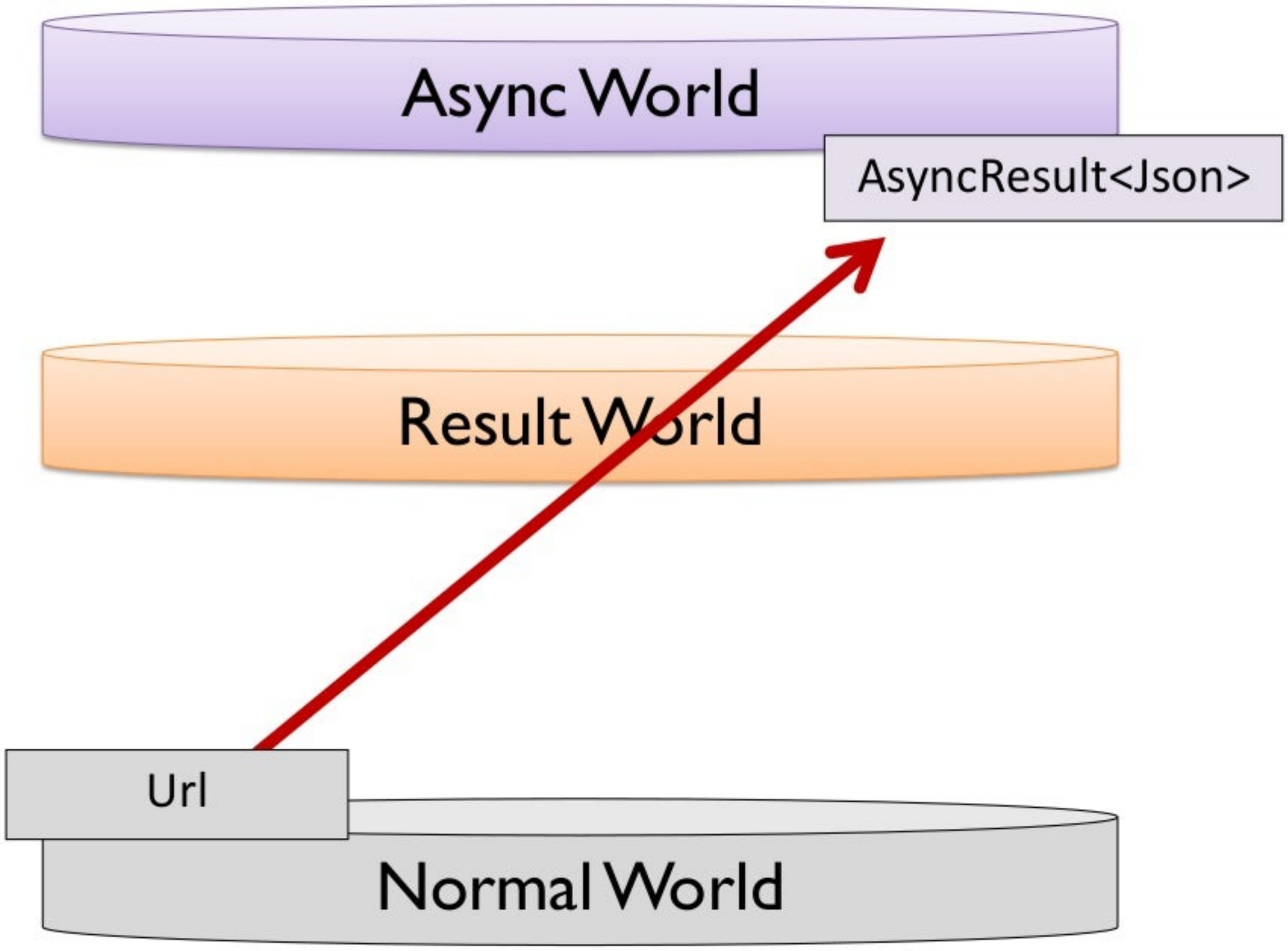
Example

Using all the tools together

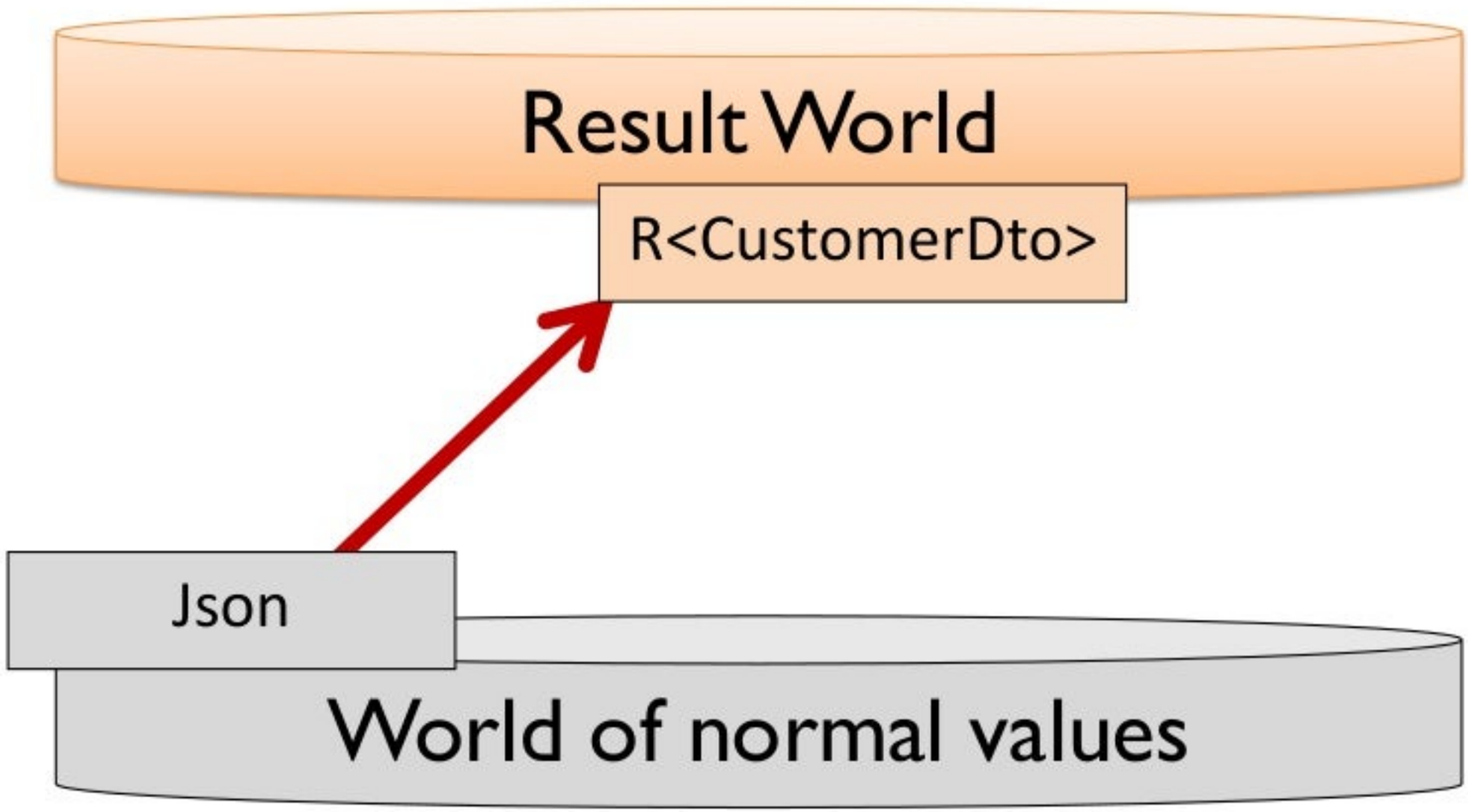
Example scenario

- 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

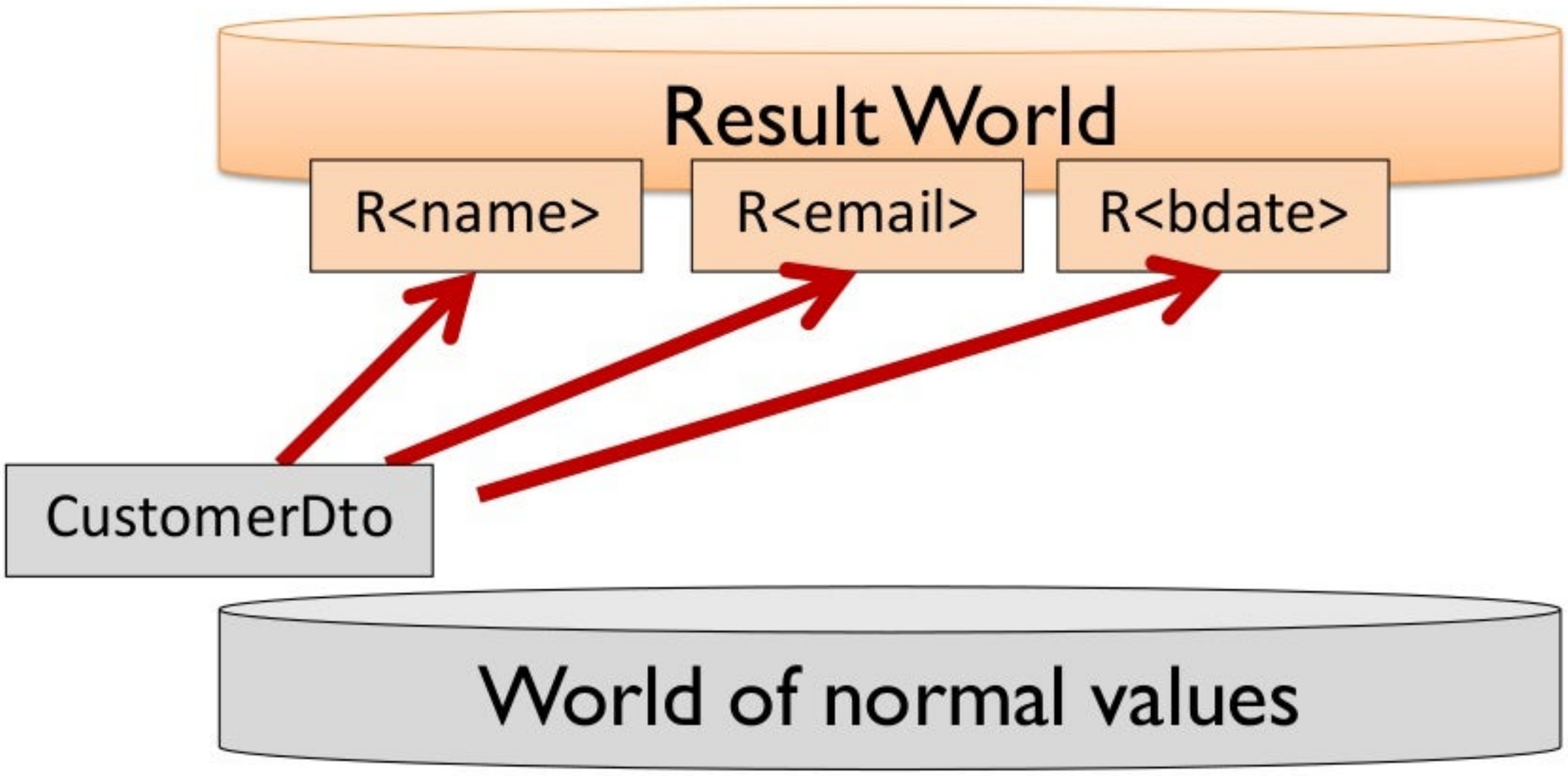
Download the json file



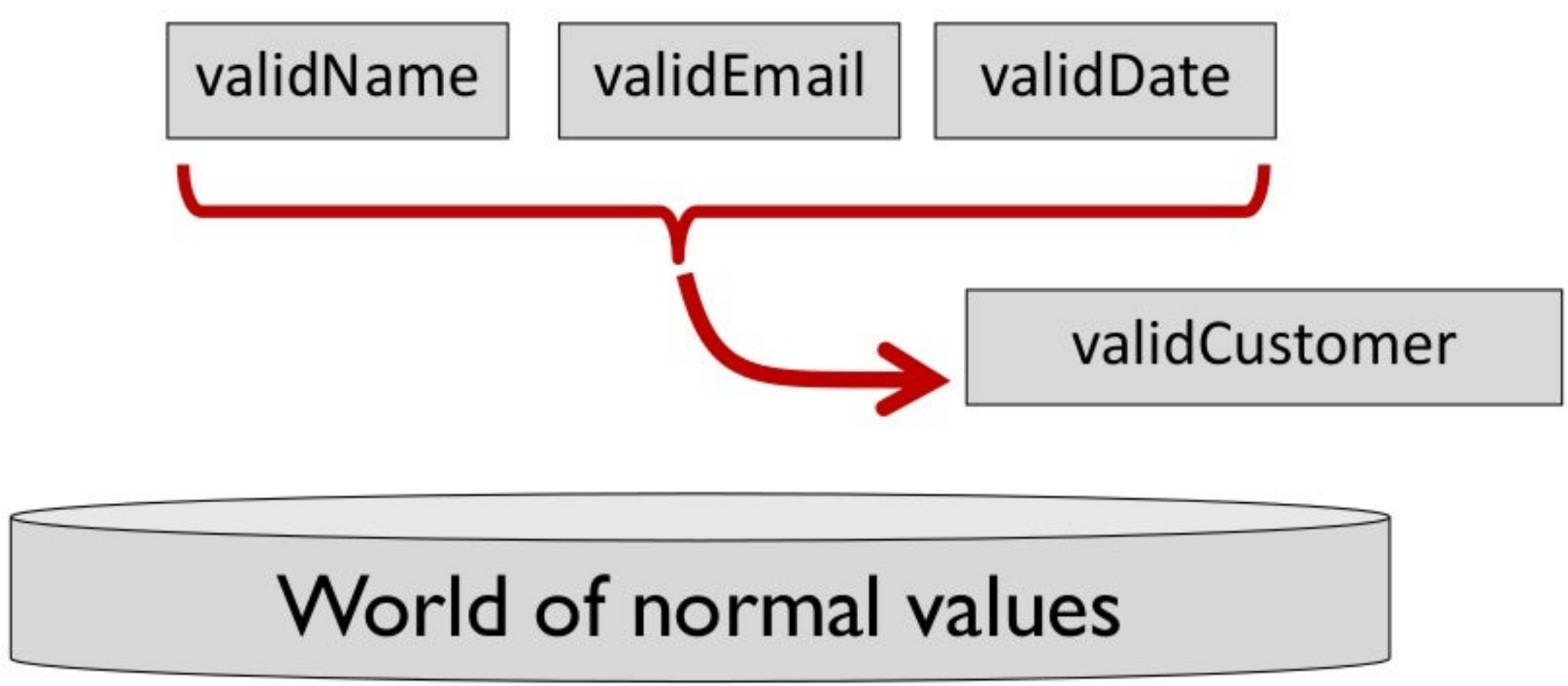
Decode the json



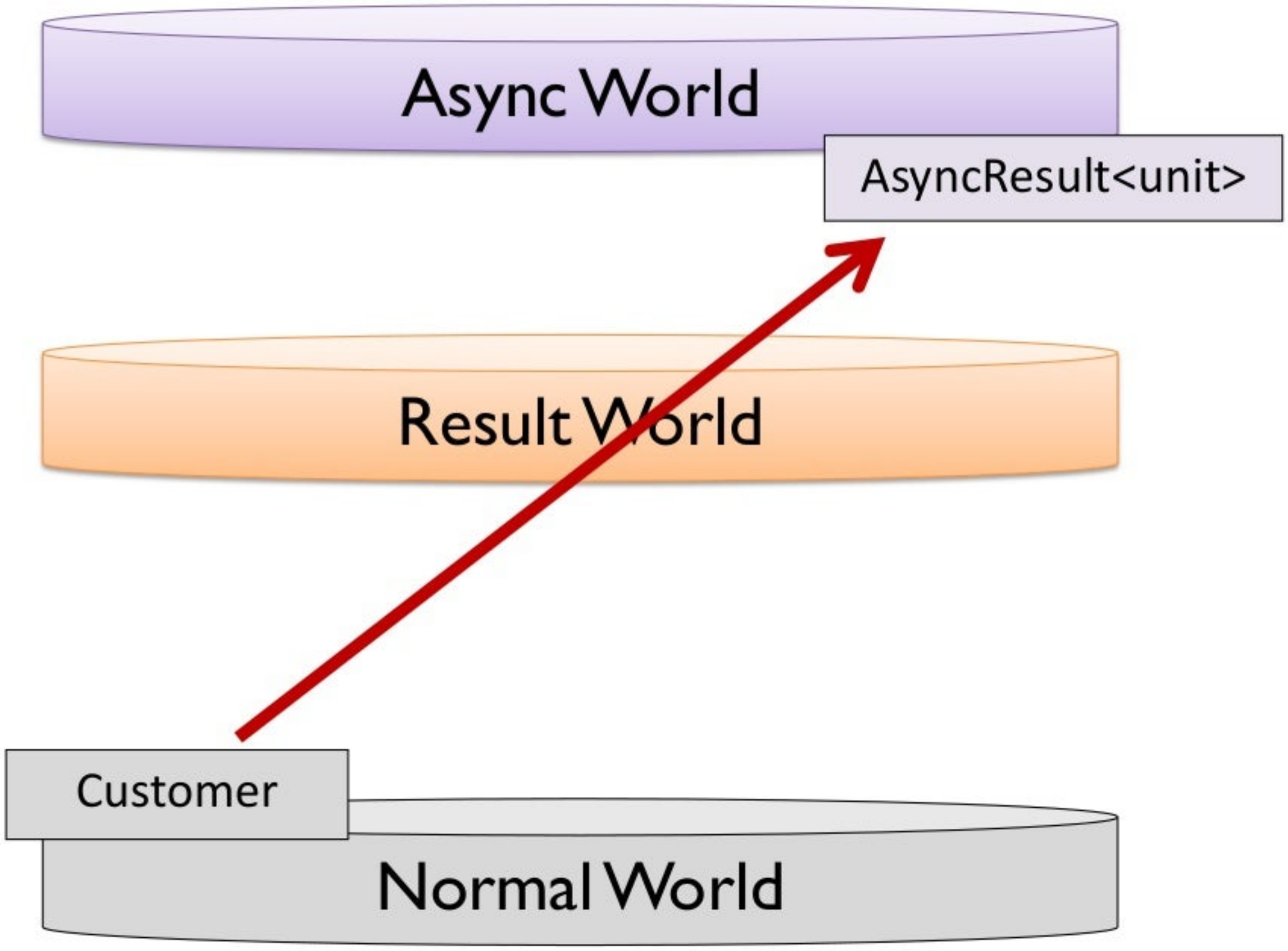
Validate fields



Construct the customer



Store the customer

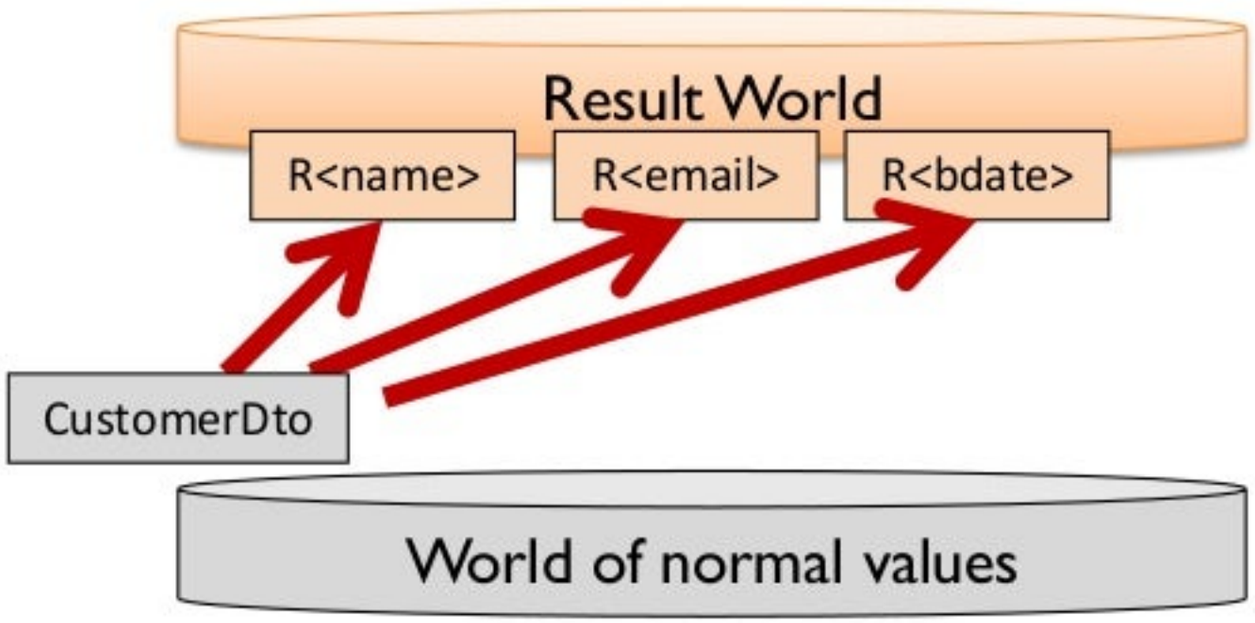


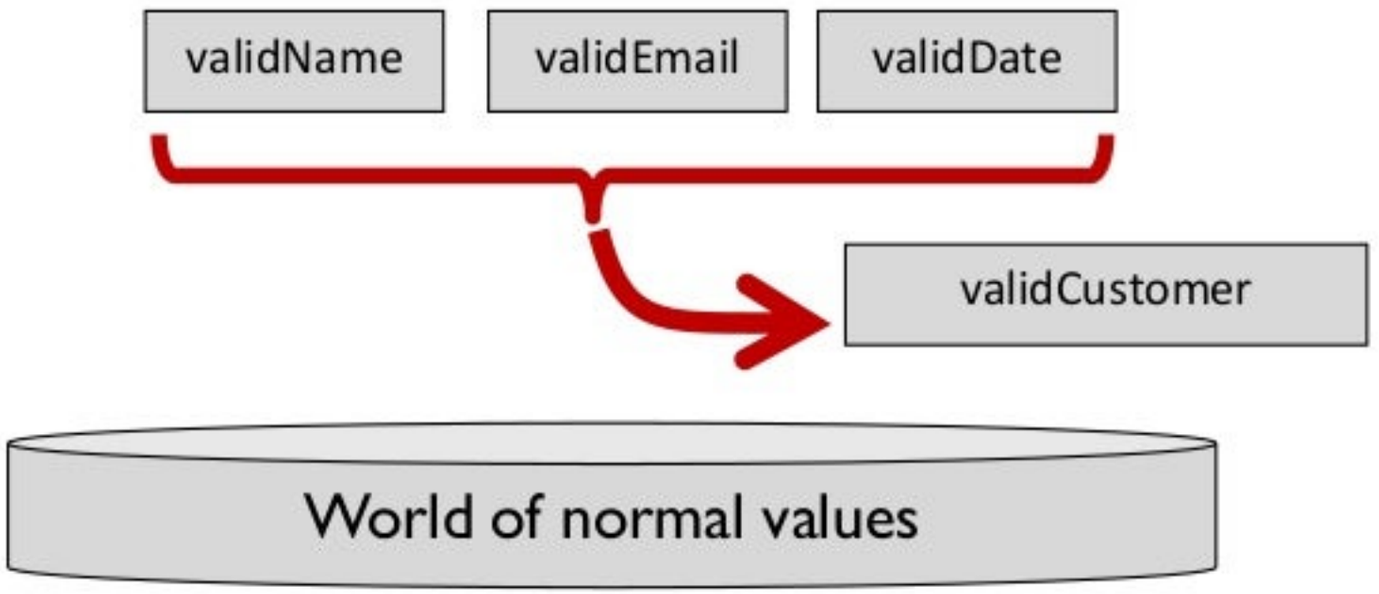
How do we compose these functions together? 😟

None the worlds match up ...
... but we can use the functional toolkit!

Validate fields AND create a customer

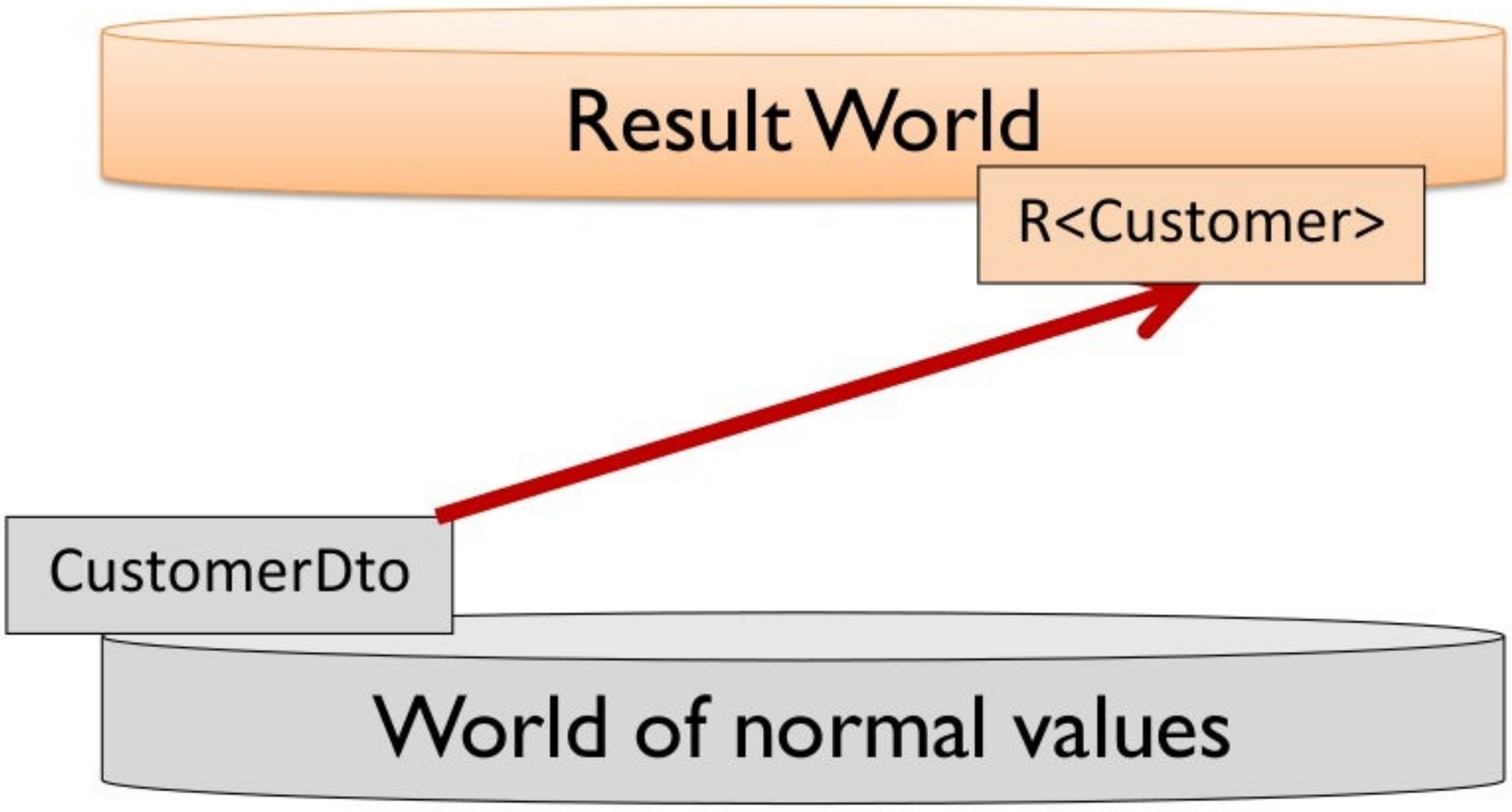
We already did this one using applicatives (and monoids)





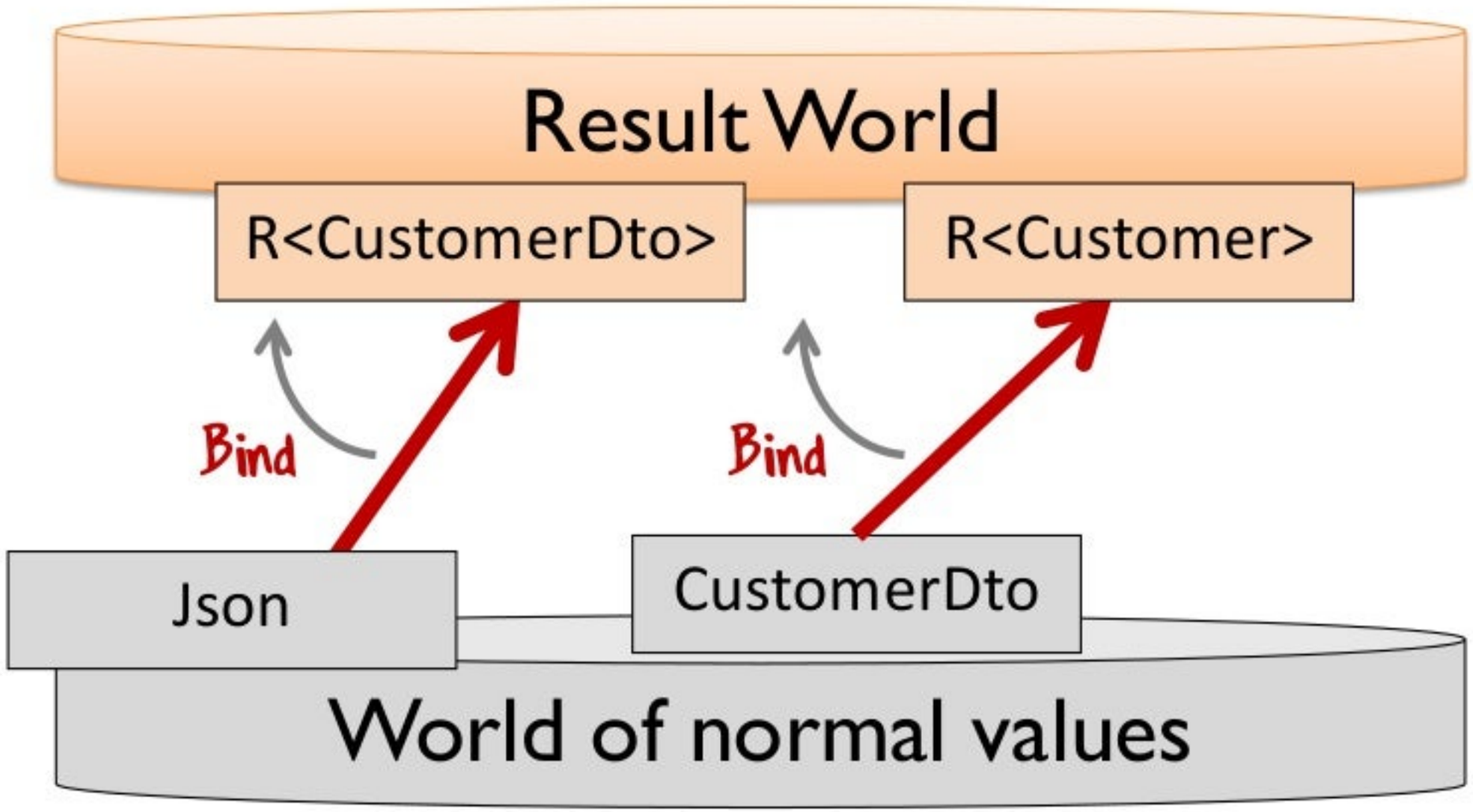
Validate fields AND create a customer

We already did this one using applicatives, and monoids

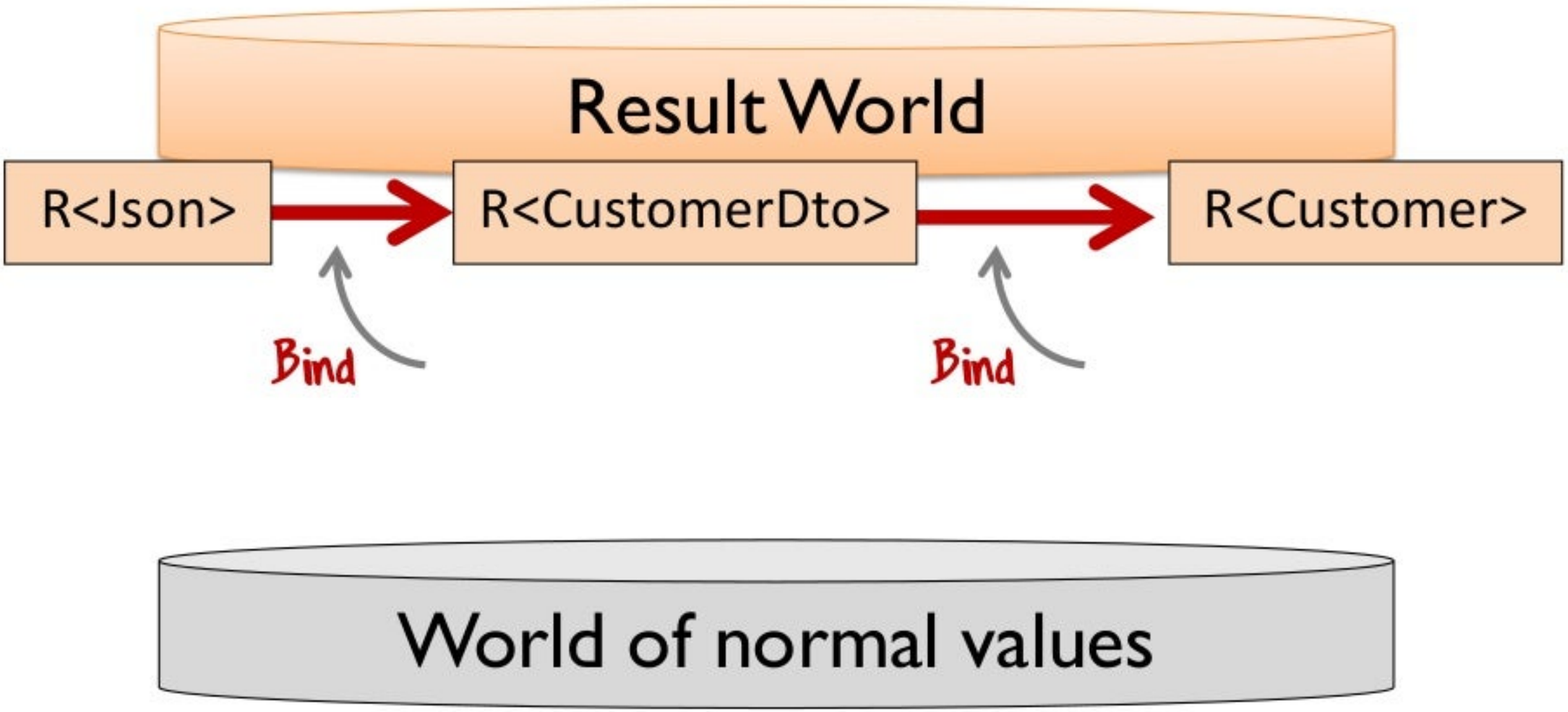


Parse json AND create a customer

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



Parse json AND create a customer



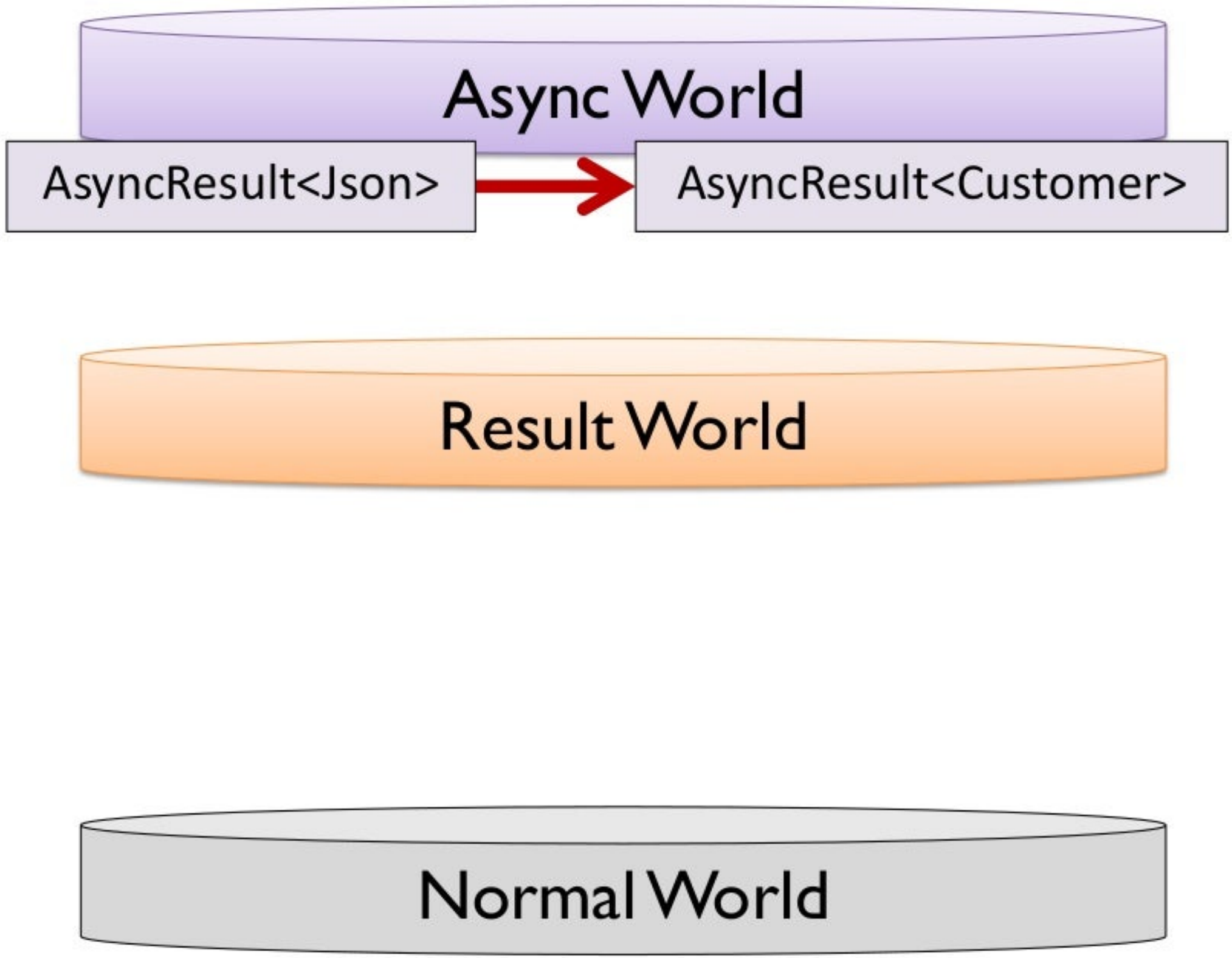
Parse json AND create a customer

What the code looks like

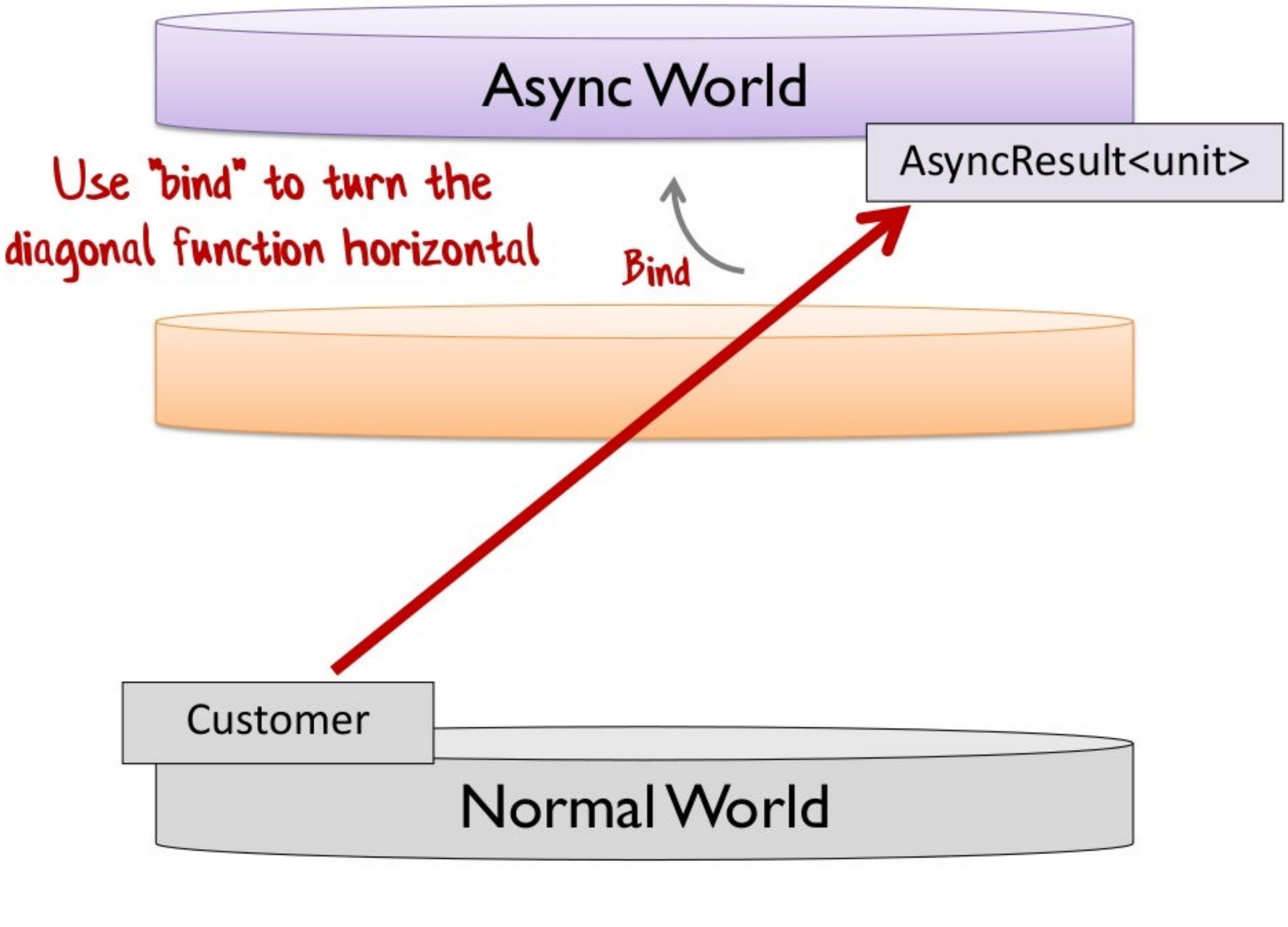
```
let processCustomerDto jsonOrError =
  jsonOrError
  |> Result.bind decodeCustomerDto
  |> Result.bind createValidCustomer
```

Parse json AND create a customer

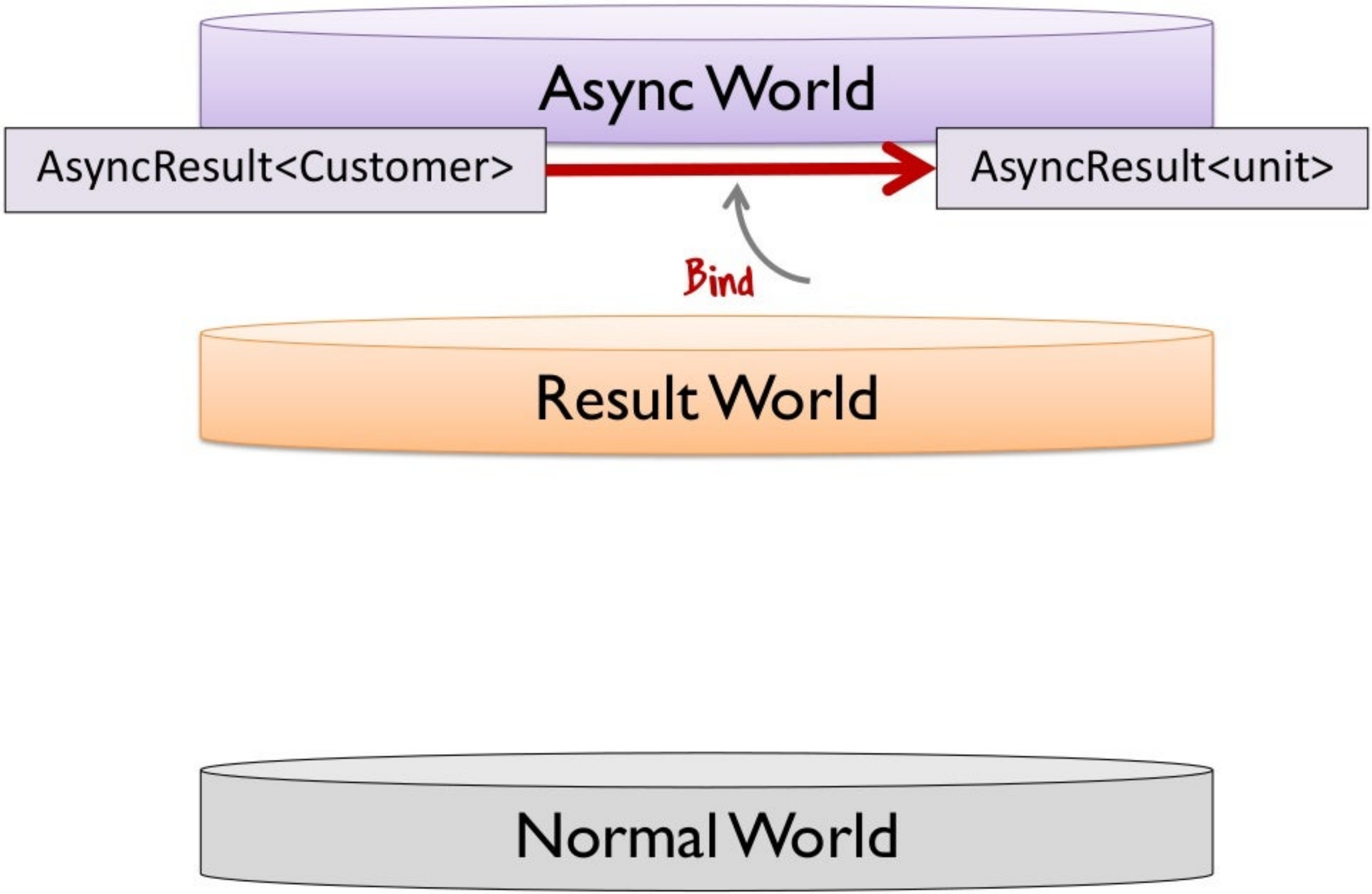
Parse json AND create a customer



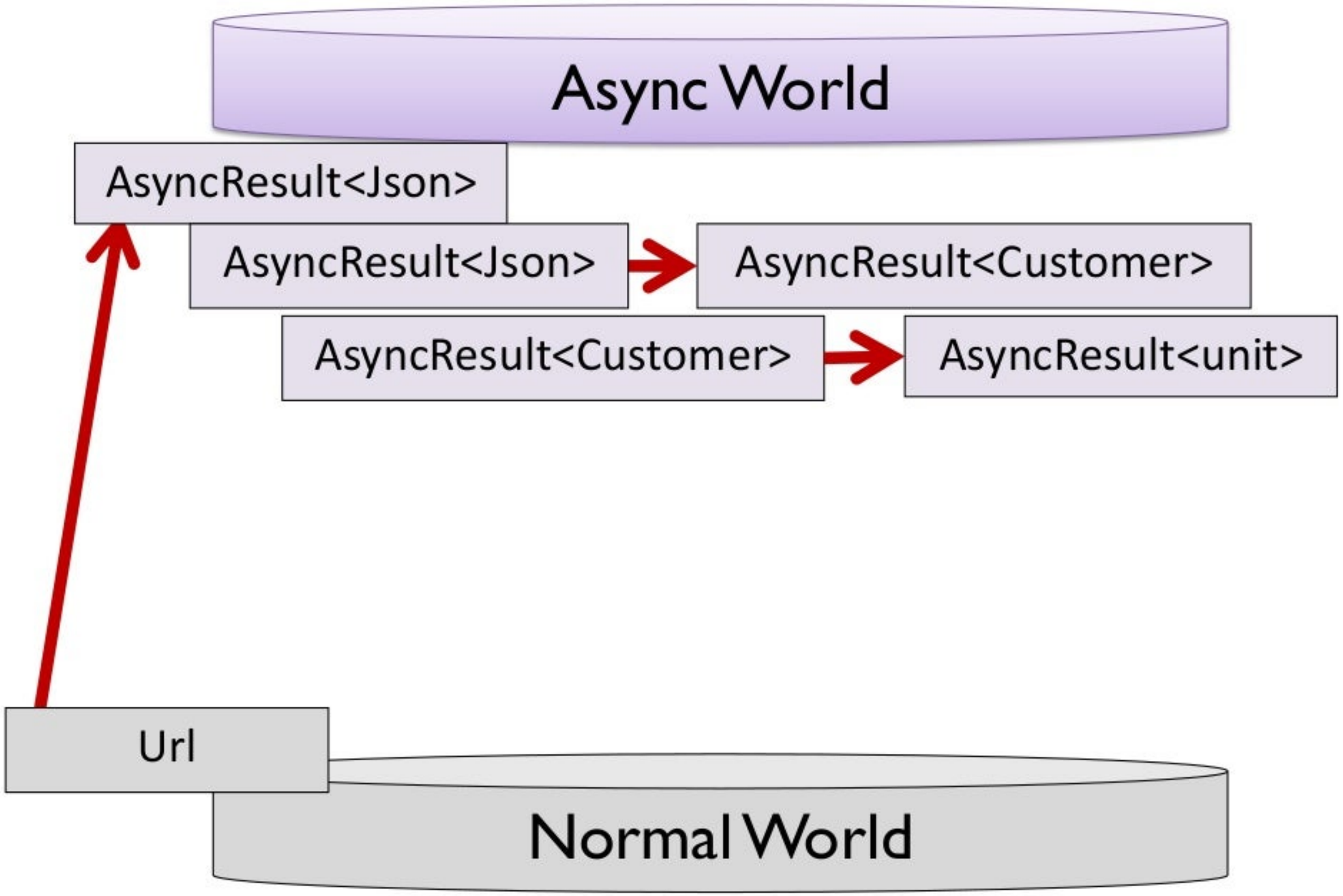
Store the customer



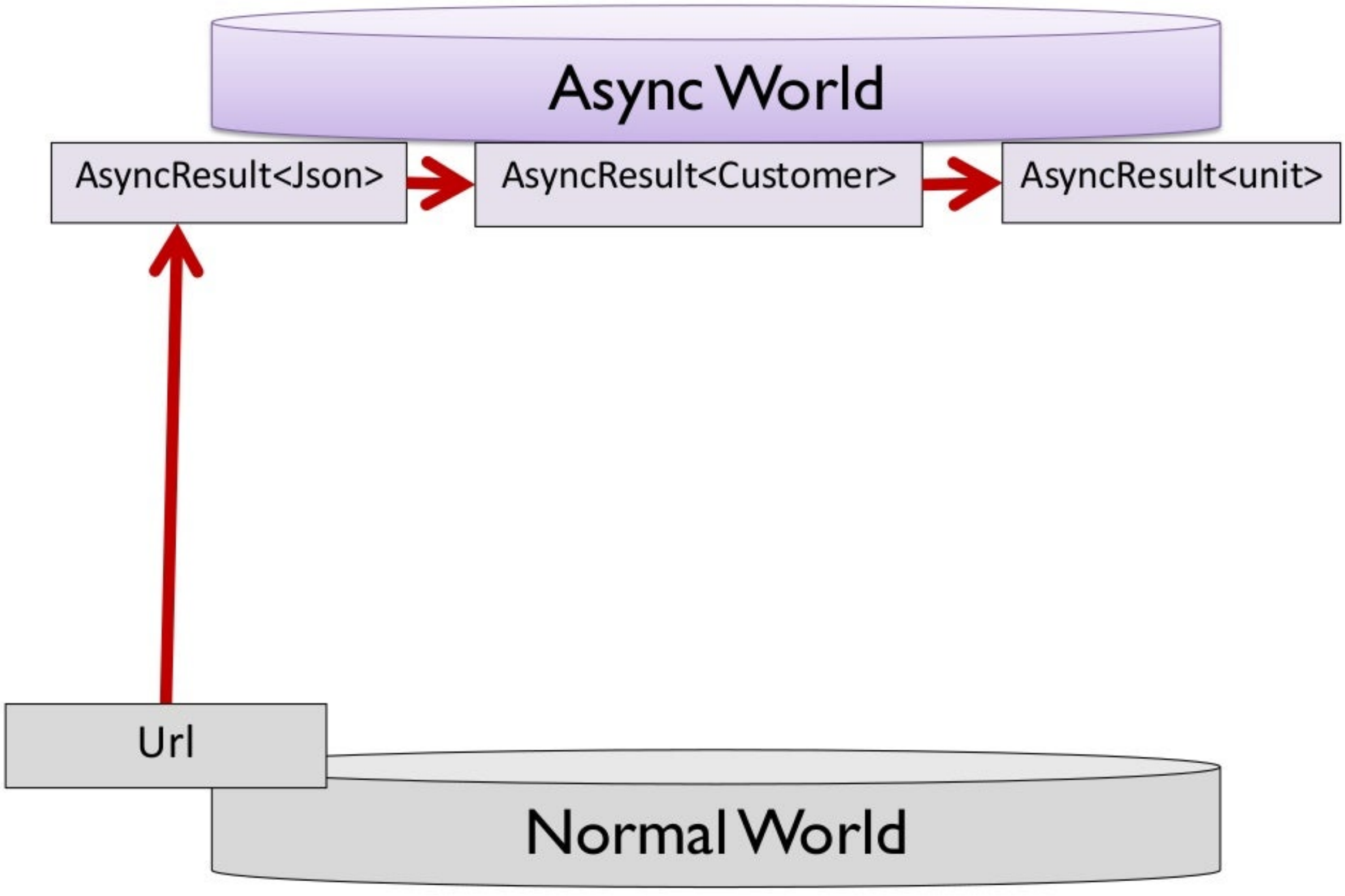
Store the customer



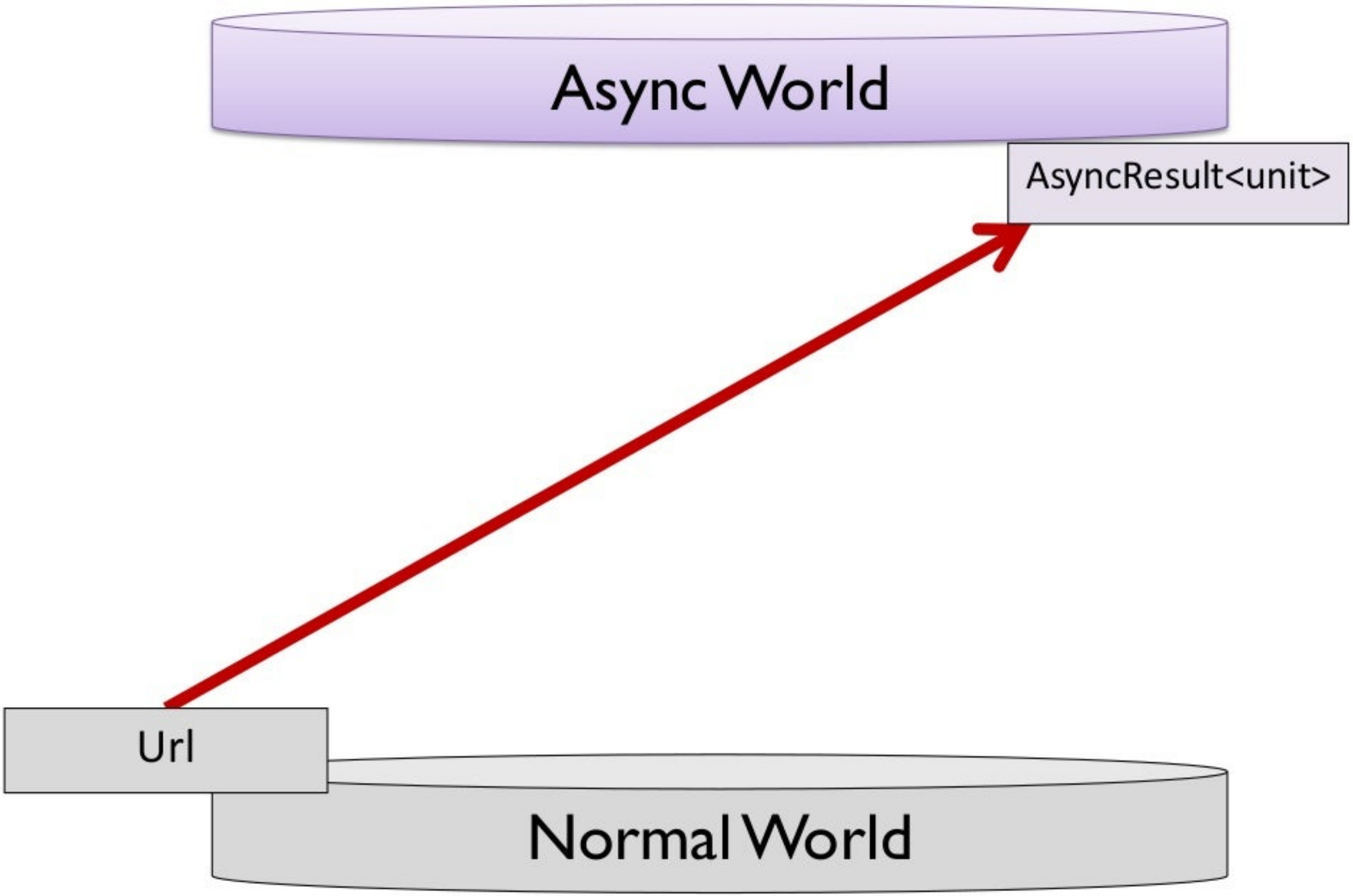
All steps are now composable



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.

"The Functional Programming Toolkit"

- Slides and video will be posted at
  - fsharpforfunandprofit.com/fptoolkit

Related talks

- "Functional Design Patterns"
  - fsharpforfunandprofit.com/fppatterns
- "The Power of Composition"
  - fsharpforfunandprofit.com/composition
- "Domain Modeling Made Functional"
  - fsharpforfunandprofit.com/ddd

Twitter: @ScottWlaschin

Thanks!