Functional Design Patterns

(NDC London 2014)

@ScottWlaschin

fsharpforfunandprofit.com

fsharpWorks

Practices?

Approaches?

Functional Design Patterns

Tips?

@ScottWlaschin

fsharpforfunandprofit.com

fsharpWorks

Functional Design Patterns

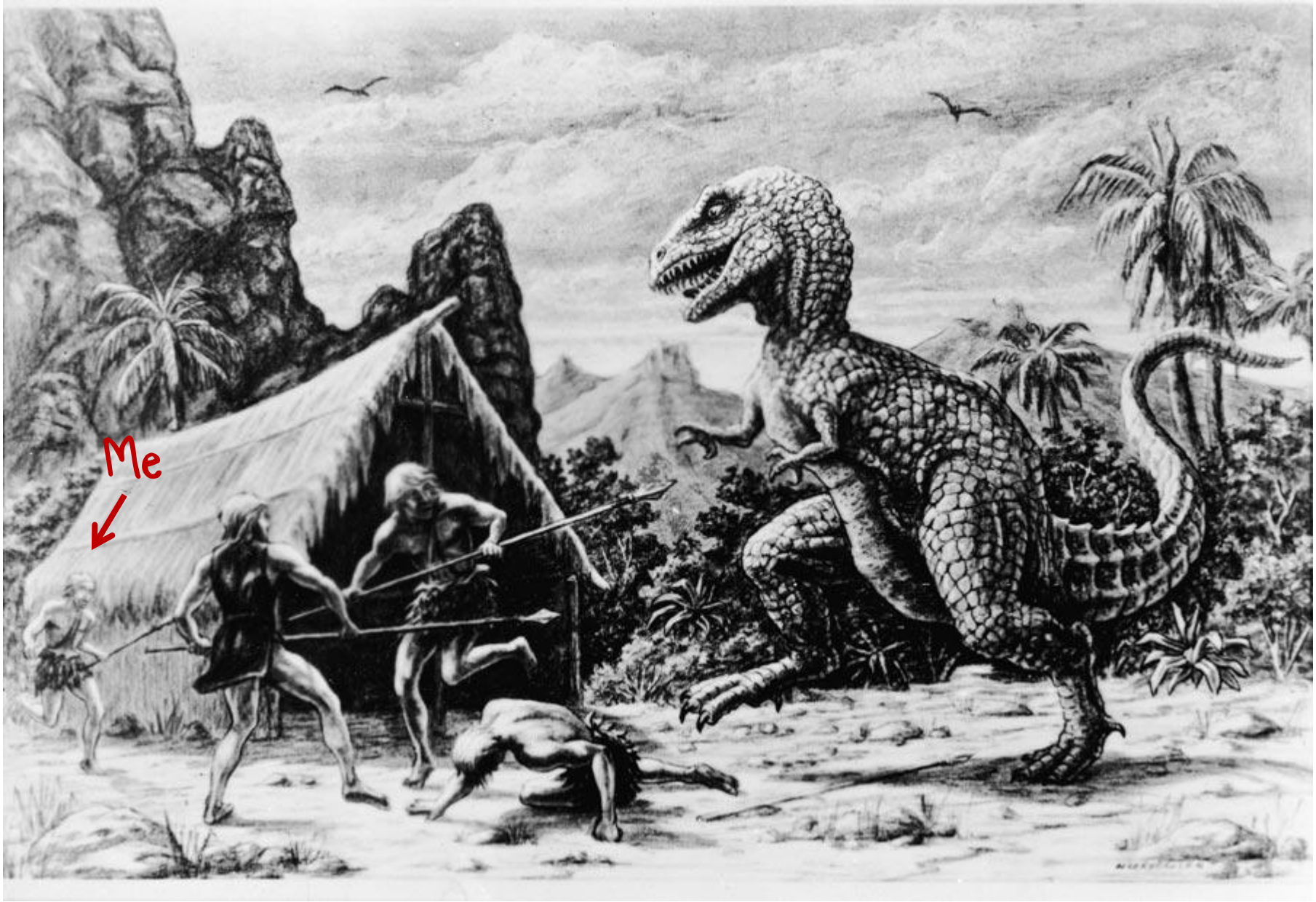
So I'll reluctantly stick with this...

@ScottWlaschin

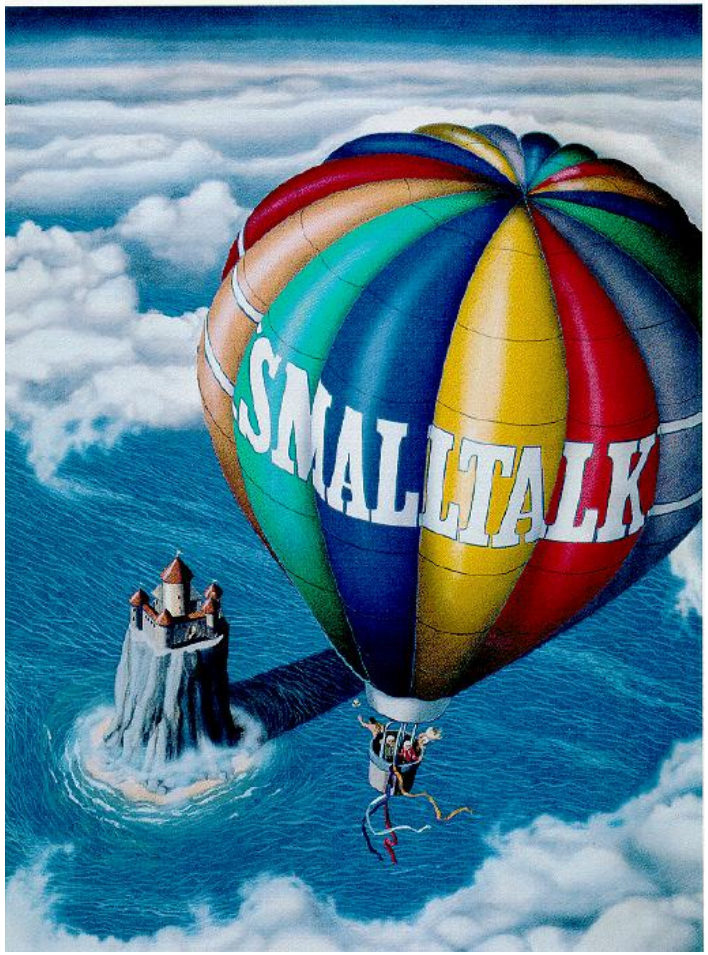
fsharpforfunandprofit.com

fsharpWorks

HOW I GOT HERE


Me

I ♥



I ♥ 

but...

Making fun of Enterprise OO
is like shooting fish in a barrel

I ♥ 

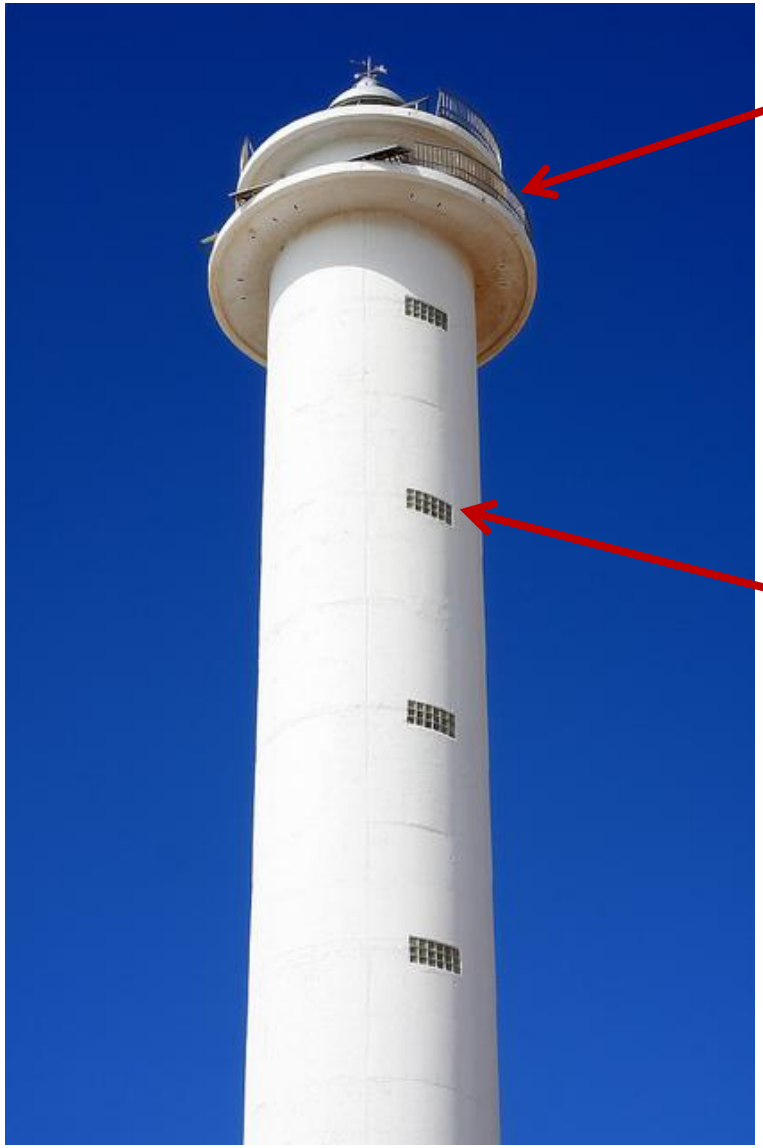
but...

Making fun of Enterprise OO
is like shooting MarineVertebrates
in an AbstractBarrelProxyFactory

Inspired by @sjkilleen

I  

This is what happens when you dabble too much in functional programming



https://www.flickr.com/photos/37511372@N08/5488671752/

FP DESIGN PATTERNS

OO pattern/principle

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

OO pattern/principle	Borg response
• Single Responsibility Principle	
• Open/Closed principle	
• Dependency Inversion Principle	
• Interface Segregation Principle	
• Factory pattern	
• Strategy pattern	
• Decorator pattern	
• Visitor pattern	

OO pattern/principle	FP equivalent
Single Responsibility Principle	Functions
Open/Closed principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory pattern	You will be assimilated!
Strategy pattern	Functions again
Decorator pattern	Functions
Visitor pattern	Resistance is futile!

Seriously, FP patterns are different

Functional patterns

- Apomorphisms
- Dynamorphisms
- Chronomorphisms
- Zygohistomorphic prepromorphisms

Functional patterns

- Core Principles of FP design
  - Functions, types, composition
- Functions as parameters
  - Functions as interfaces
  - Partial application & dependency injection
  - Continuations, chaining & the pyramid of doom
- Monads
  - Error handling, Async
- Maps
  - Dealing with wrapped data
  - Functors
- Monoids
  - Aggregating data and operations

This talk

A whirlwind tour of many sights

Don't worry if you don't understand everything

Important to understand!

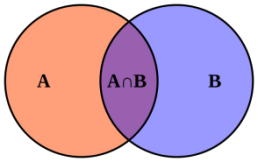
(SOME) CORE PRINCIPLES OF FUNCTIONAL PROGRAMMING

Core principles of FP

Functions are things

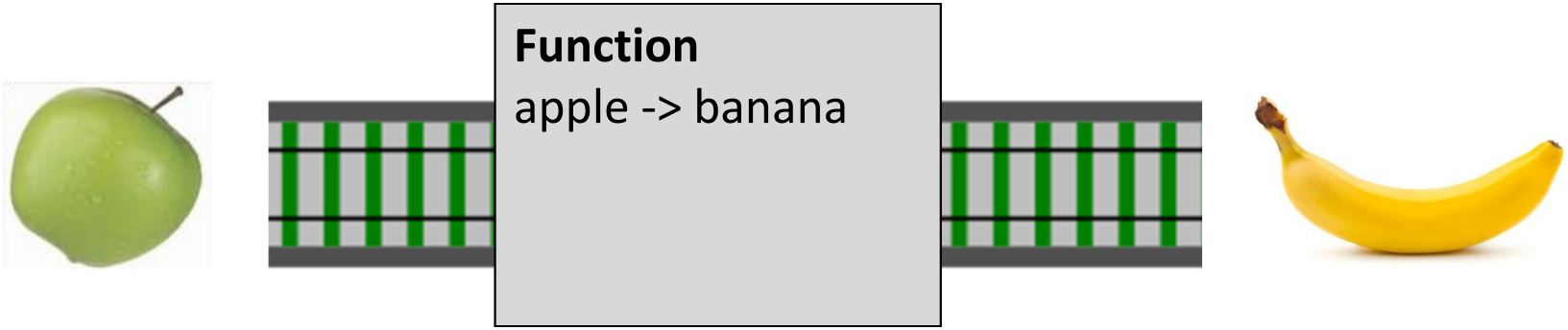
Composition everywhere

Types are not classes



Core principle:
Functions are things

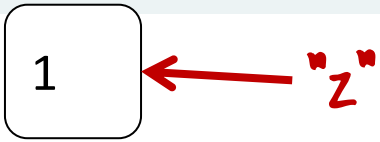
Functions as things



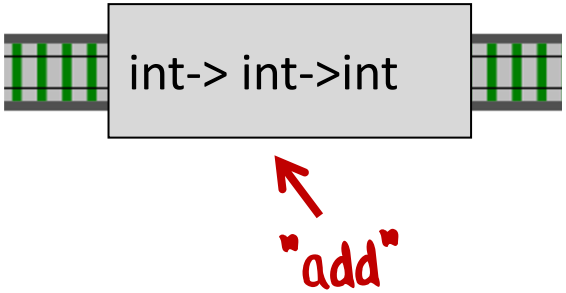
A function is a standalone thing,
not attached to a class

Functions as things

```
let z = 1
```



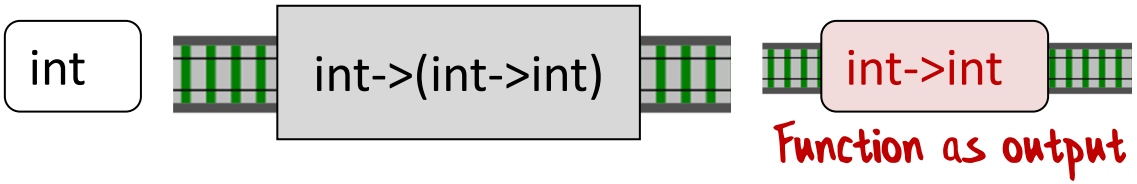

```
let add x y = x + y
```



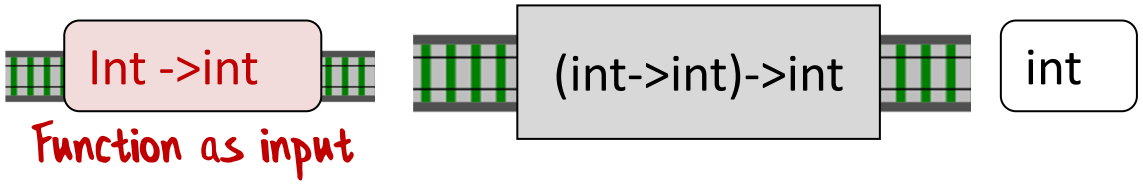
Same keyword
(not a coincidence)

Functions as inputs and outputs

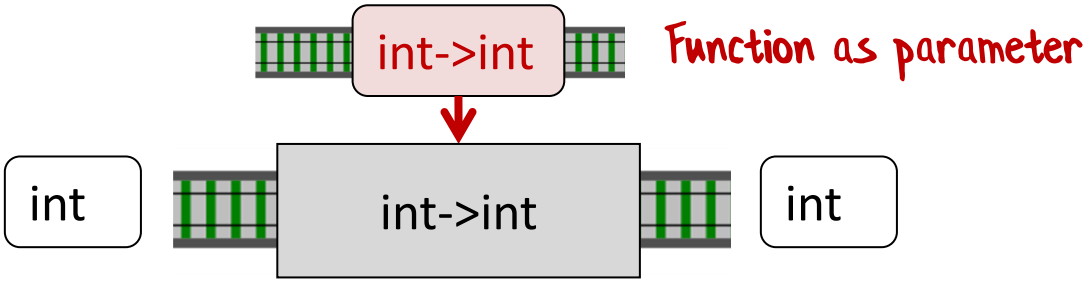
```
let add x = (fun y -> x + y)
```




```
let useFn f = (f 1) + 2
```



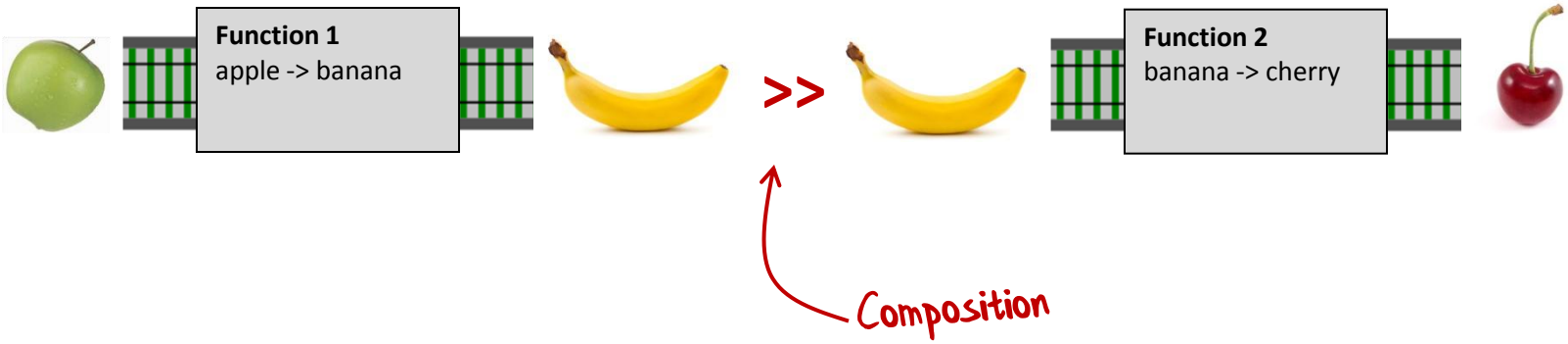

```
let transformInt f x = (f x) + 1
```



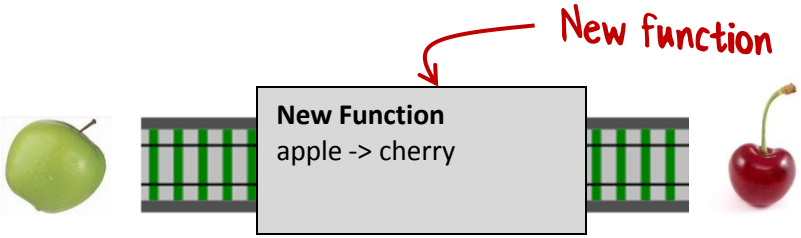
Core principle:
Composition everywhere

Function composition

Function composition



Function composition



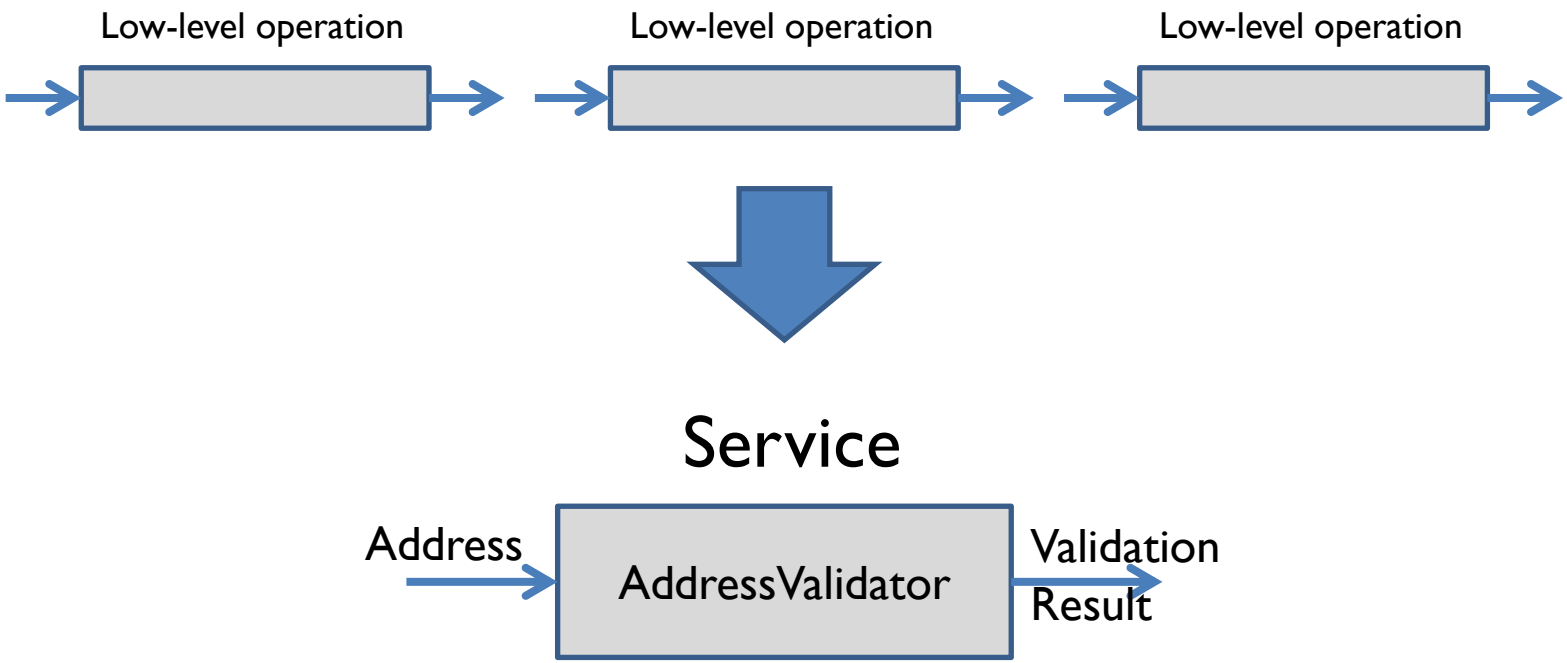
Can't tell it was built from smaller functions!

Design paradigm:
Functions all the way down

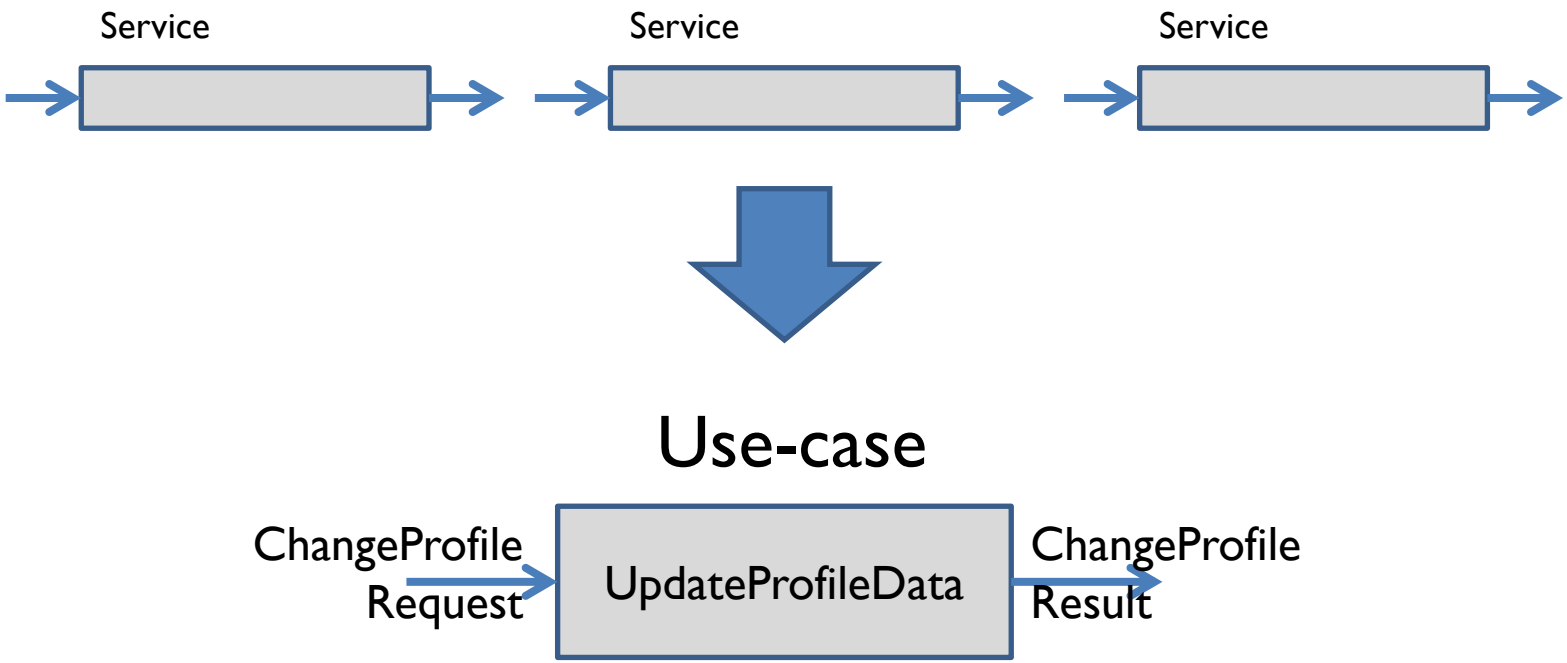
~~"Functions in the small,
objects in the large"~~

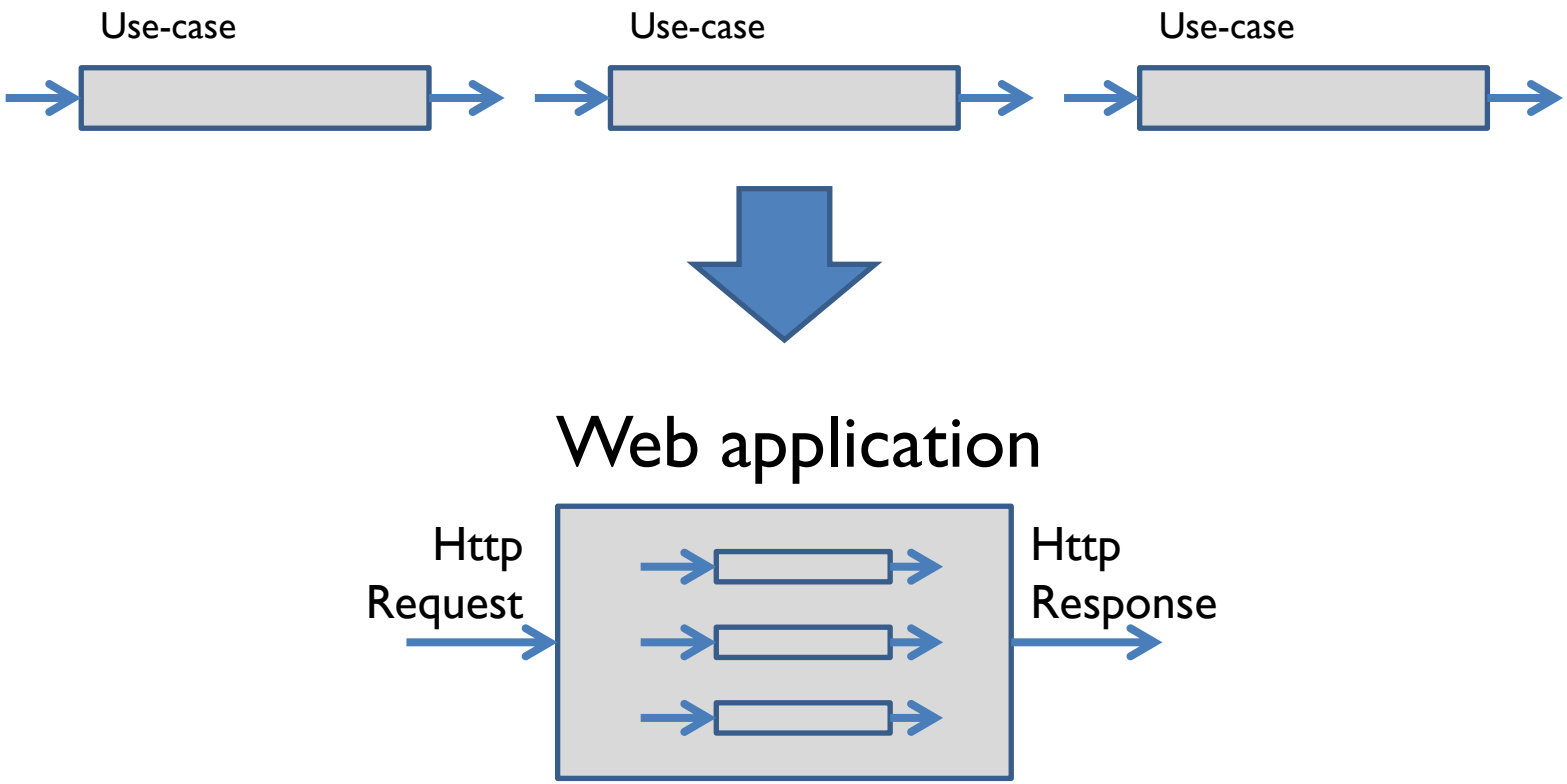
"Functions in the small,
functions in the large"

Low-level operation



A "Service" is just like a microservice
but without the "micro" in front

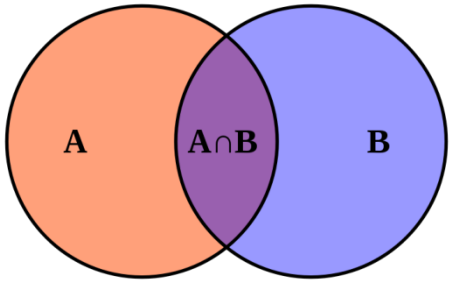

Service
Service
Service
Use-case
ChangeProfile
Request
UpdateProfileData
ChangeProfile
Result

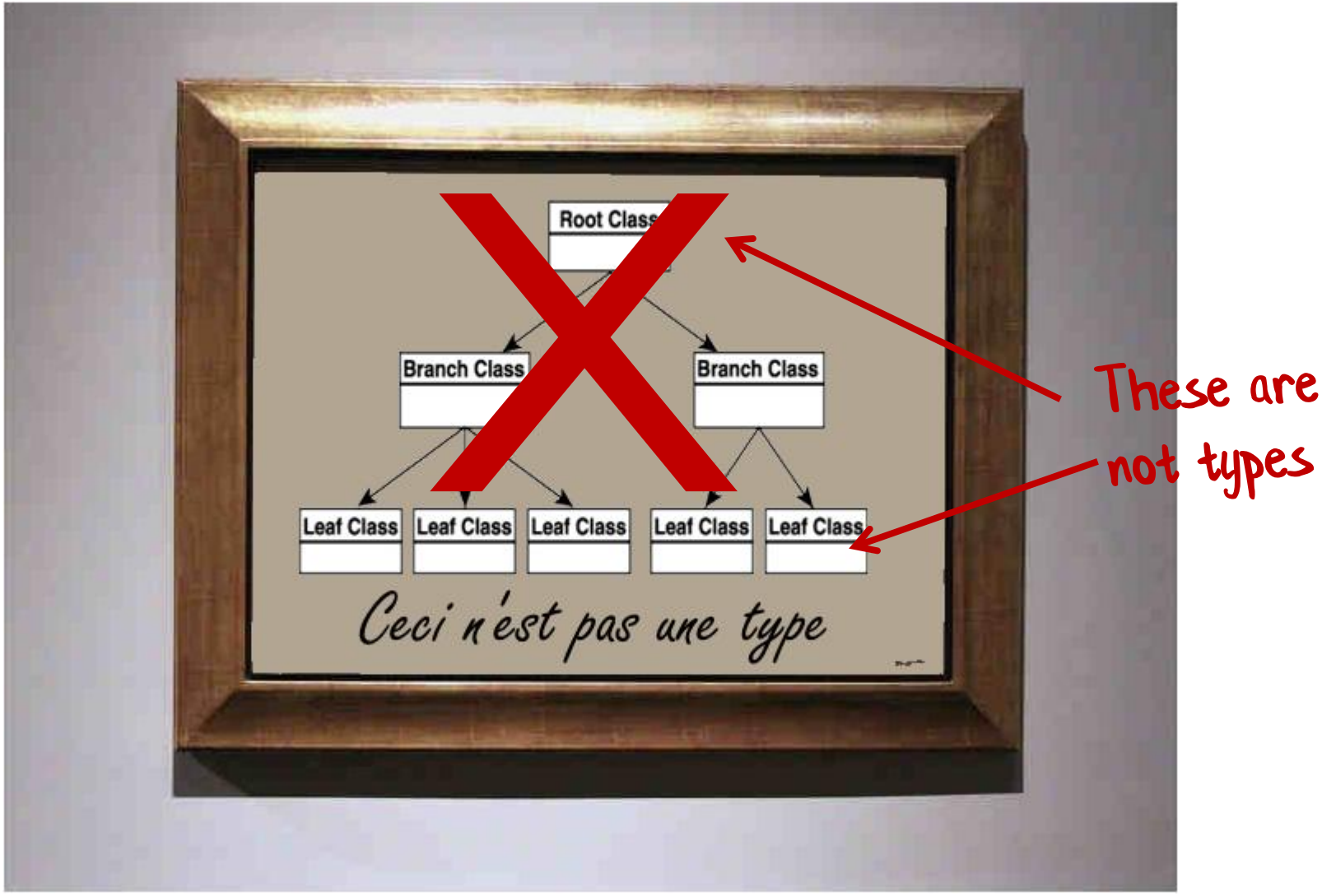


"Composition is fractal"

Core principle:

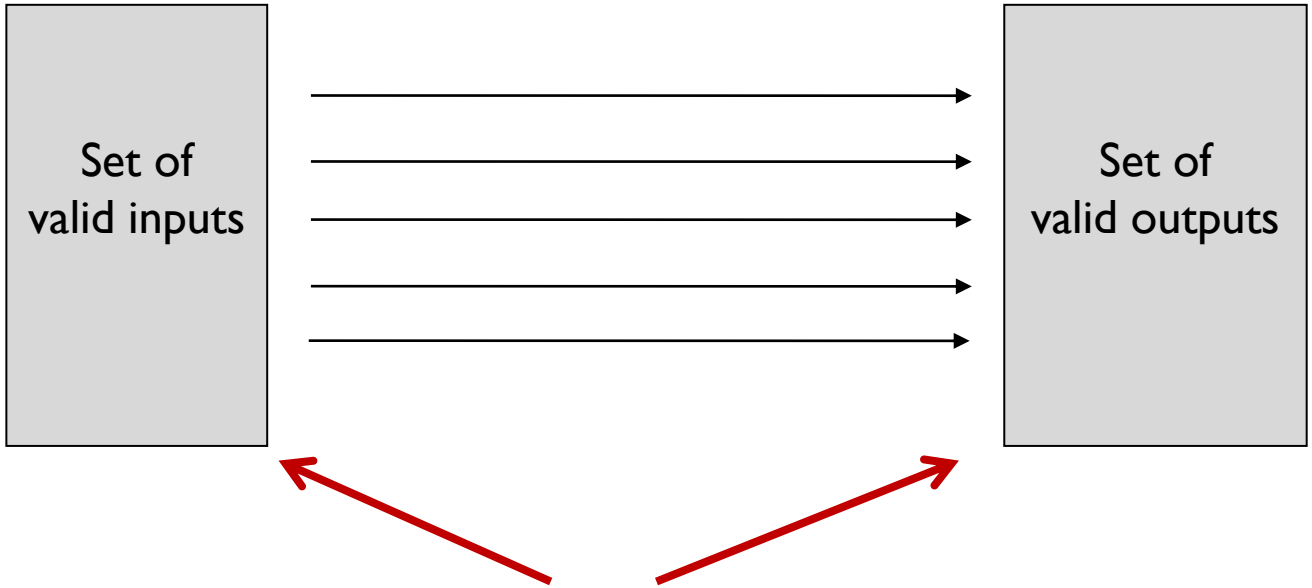
Types are not classes




Root Class
Branch Class
Branch Class
Leaf Class
Leaf Class
Leaf Class
Leaf Class
Leaf Class
Ceci n'est pas une type
These are
not types

Types are not classes

So what is a type?



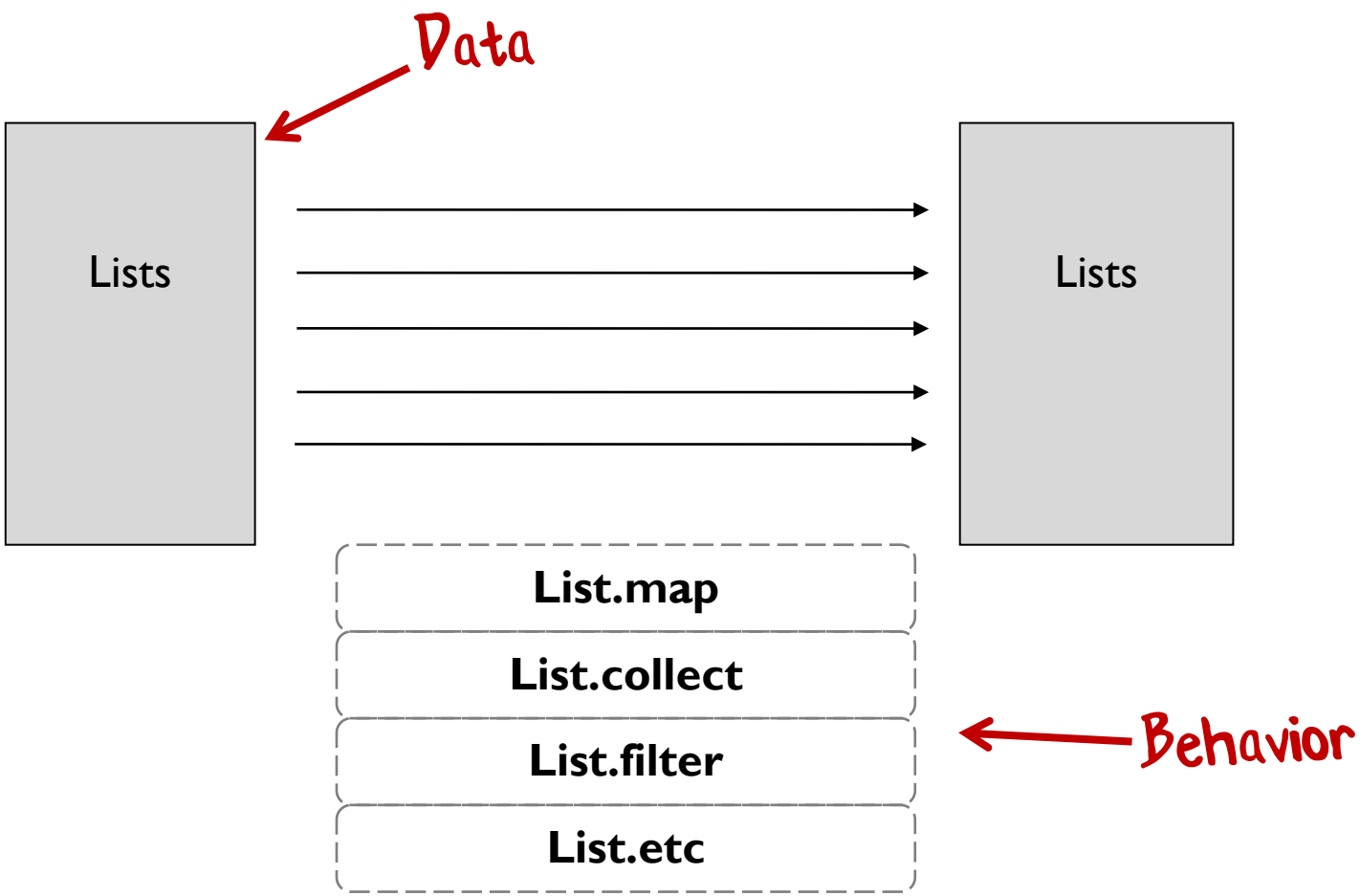
These are types

"Int" is a type

"Customer" is a type

"int->int" is a type

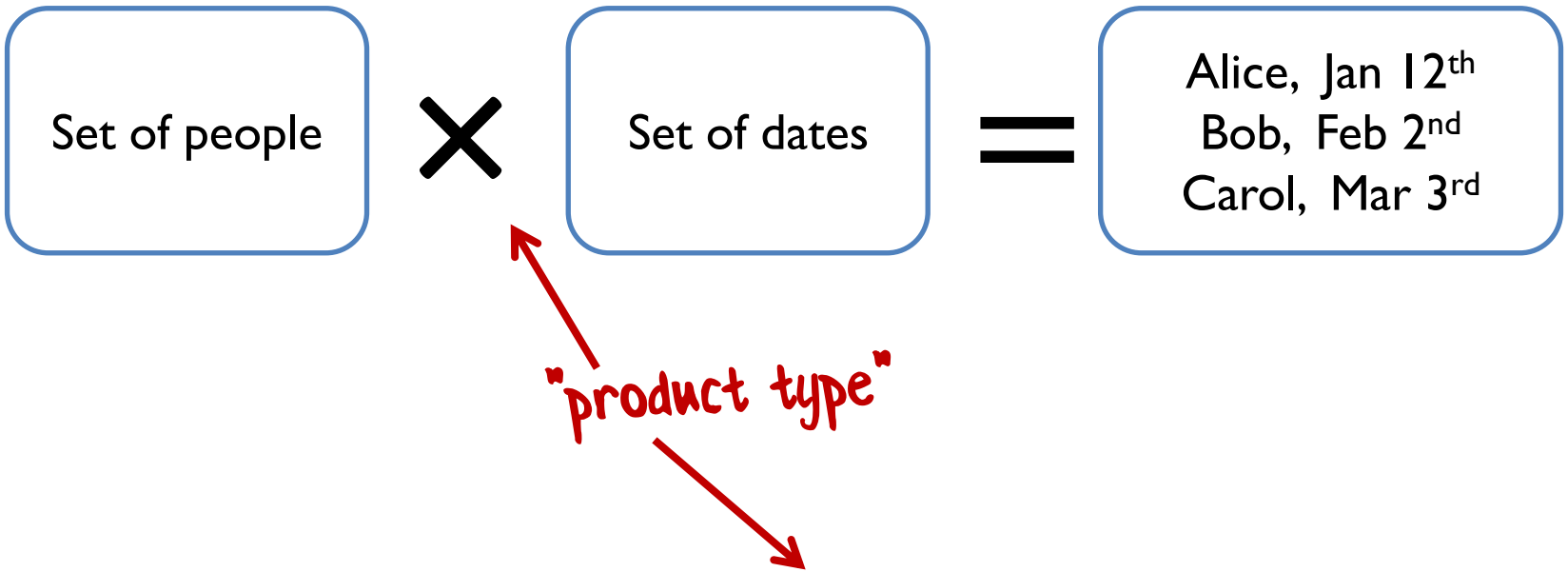
Types separate data from behavior



Types can be composed too

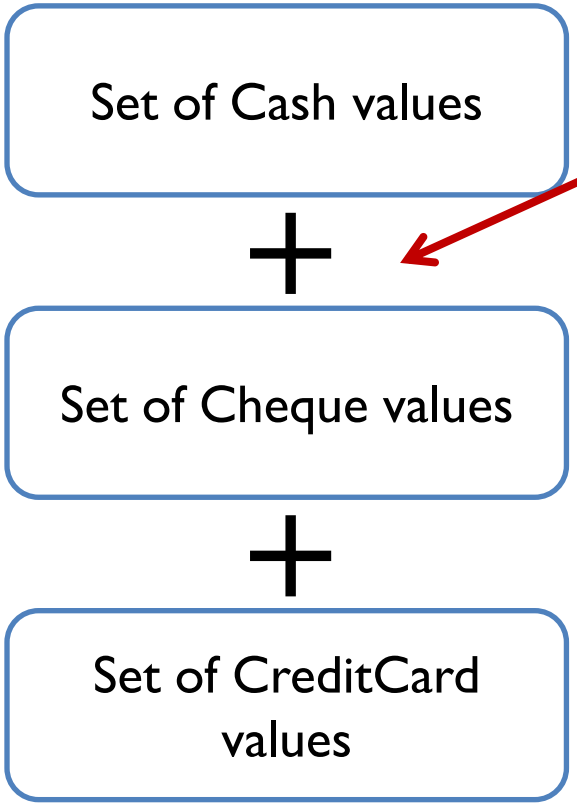
"algebraic types"

Product types



type **Birthday** = Person * Date

Sum types



"sum type"

```
type PaymentMethod =
  | Cash
  | Cheque of ChequeNumber
  | Card of CardType * CardNumber
```

Design principle:
Strive for totality

Totality

Domain (int)

…
3
2
1
0
…

```
int TwelveDividedBy(int input)
{
    switch (input)
    {
    case 3: return 4;
    case 2: return 6;
    case 1: return 12;
    case 0: return ??;
    }
}
```

Codomain (int)

…
4
6
12
…

Totality

Domain (int)

…
3
2
1
0
…

```
int TwelveDividedBy(int input)
{
    switch (input)
    {
    case 3: return 4;
    case 2: return 6;
    case 1: return 12;
    case 0: return ??;
    }
}
```

Codomain (int)

…
4
6
12
…

What happens here?

Totality

Domain (int)

…
3
2
1
0
…

```
int TwelveDividedBy(int input)
{
    switch (input)
    {
    case 3: return 4;
    case 2: return 6;
    case 1: return 12;
    case 0:
     throw InvalidArgException;
    }
}
```

Codomain (int)

…
4
6
12
…

You tell me you can handle Ø, and then you complain about it?

int -> int

This type signature is a lie!

Totality

Constrain the input

NonZeroInteger

…
3
2
1
-1
…

```
int TwelveDividedBy(int input)
{
    switch (input)
    {
    case 3: return 4;
    case 2: return 6;
    case 1: return 12;

    case -1: return -12;
    }
}
```

int

…
4
6
12
…

∅ is missing

∅ doesn't have to be handled

NonZeroInteger -> int

Types are documentation

Totality

Extend the output

int

…
3
2
1
0
-1
…

```
int TwelveDividedBy(int input)
{
    switch (input)
    {
    case 3: return Some 4;
    case 2: return Some 6;
    case 1: return Some 12;
    case 0: return None;
    }
}
```

Option<Int>

…
Some 4
Some 6
Some 12
None
…

0 is valid input

int -> int option

Types are documentation

Design principle:
Use static types for domain modelling and documentation

Static types only!
Sorry Clojure and JS developers ☹

Big topic! Not enough time today ☹!

More on DDD and designing with types at
fsharpforfunandprofit.com/ddd

FUNCTIONS AS PARAMETERS

Guideline:

Parameterize all the things

Parameterize all the things

Hard-coded data. Yuck!

```
let printList() =
   for i in [1..10] do
      printfn "the number is %i" i
```

Parameterize all the things

It's second nature to parameterize the data input:

```
let printList aList =
   for i in aList do
      printfn "the number is %i" i
```

Hard-coded behaviour. Yuck!

Parameterize all the things

FPers would parameterize the action as well:

```
let printList anAction aList =
   for i in aList do
      anAction i
```

We've decoupled the behavior from the data. Any list, any action!

A good language helps by making this trivial to do!

Parameterize all the things

```
public static int Product(int n)
{
    int product = 1;
    for (int i = 1; i <= n; i++)
    {
        product *= i;
    }
    return product;
}

public static int Sum(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
    {
        sum += i;
    }
    return sum;
}
```

Don't Repeat Yourself

Parameterize all the things

```
public static int Product(int n)
{
    int product = 1;
    for (int i = 1; i <= n; i++)
    {
        product *= i;
    }
    return product;
}

public static int Sum(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
    {
        sum += i;
    }
    return sum;
}
```

Common Code

Initial Value

Action

Parameterize all the things

```
let product n =
    let initialValue = 1
    let action productSoFar x = productSoFar * x
    [1..n] |> List.fold action initialValue


let sum n =
    let initialValue = 0
    let action sumSoFar x = sumSoFar+x
    [1..n] |> List.fold action initialValue
```

Initial Value

Parameterized action

Common code extracted

Lots of collection functions like this: "fold", "map", "reduce", "collect", etc.

Tip:
Function types are "interfaces"

Function types provide instant abstraction!

Function types are interfaces

```
interface IBunchOfStuff
{
   int DoSomething(int x);
   string DoSomethingElse(int x);
   void DoAThirdThing(string x);
}
```

Let's take the
Single Responsibility Principle and the
Interface Segregation Principle
to the extreme...

Every interface should have
only one method!

Function types are interfaces

```
interface IBunchOfStuff
{
   int DoSomething(int x);
}
```

An interface with one method is a just a function type

```
type IBunchOfStuff: int -> int
```

Any function with that type is compatible with it

```
let add2 x = x + 2          // int -> int
let times3 x = x * 3        // int -> int
```

Strategy pattern

Object-oriented strategy pattern:

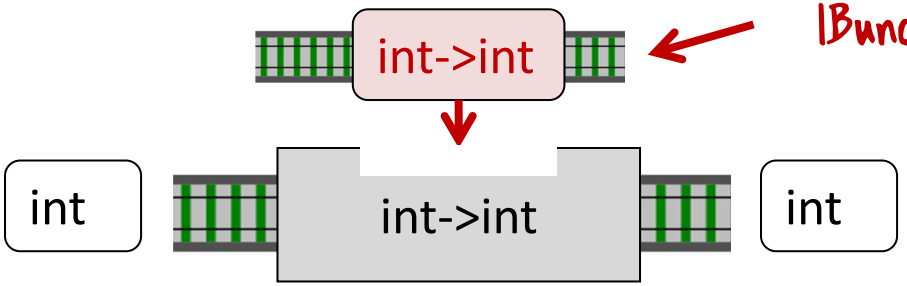
```
class MyClass
{
   public MyClass(IBunchOfStuff strategy) {..}

   int DoSomethingWithStuff(int x)
   {
      return _strategy.DoSomething(x)
   }
}
```

Strategy pattern

Functional strategy pattern:

```
let DoSomethingWithStuff strategy x =
   strategy x
```

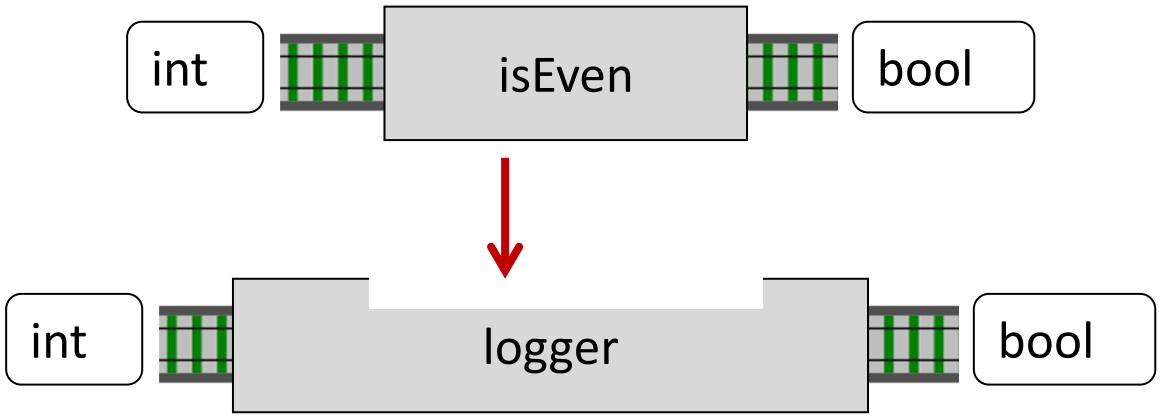
IBunchOfStuff as parameter



+ with FP approach => you don't need to create an int->int interface in advance.
& you can substitute ANY int->int function in later

Decorator pattern using function parameter:

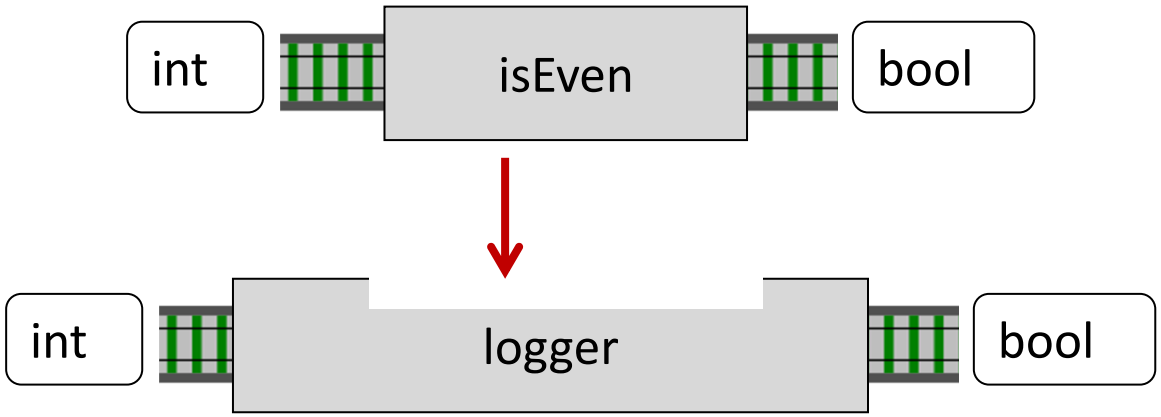
```
let isEven x = (x % 2 = 0)    // int -> bool
```




```
let logger f input =
   let output = f input
   // then log input and output
   // return output
```

Decorator pattern using function parameter:

```
let isEven x = (x % 2 = 0)    // int -> bool
```

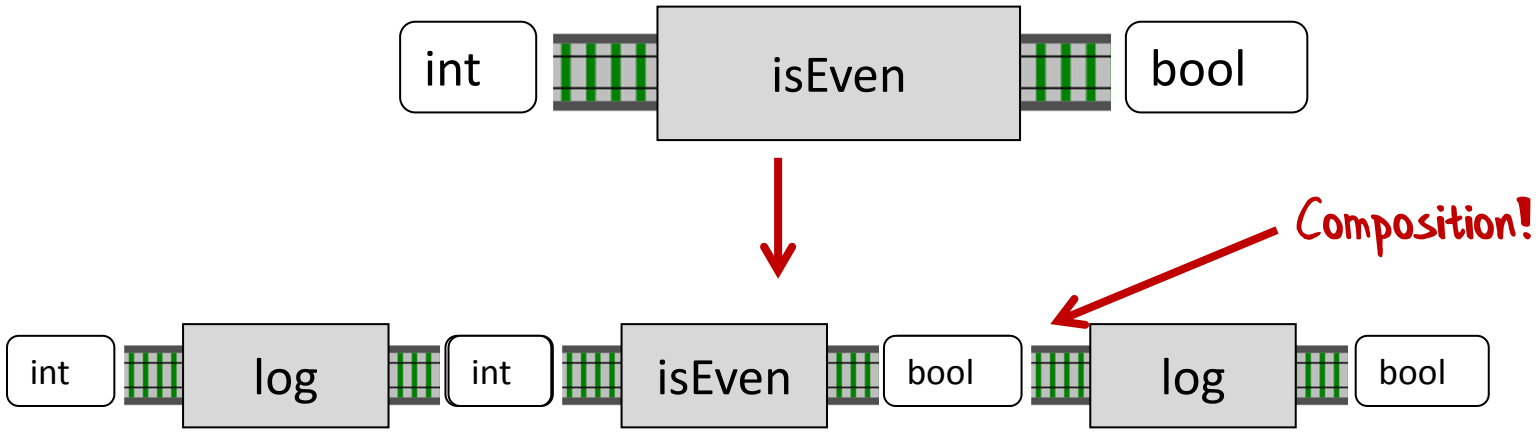



```
let isEvenWithLogging = logger isEven
                        // int -> bool
```

Substitutable for original isEven

Decorator pattern using function composition:

```
let isEven x = (x % 2 = 0)     // int -> bool
```




```
let isEvenWithLogging = log >> isEven >> log
                        // int -> bool
```

Substitutable for original isEven

Bad news:
Composition patterns
only work for functions that
have one parameter! ☹

Good news!

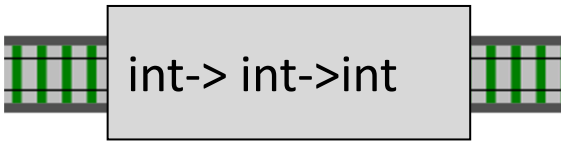
<u>Every</u> function is a

one parameter function ☺

Writing functions in different ways

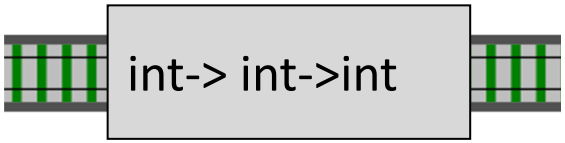
Normal (Two parameters)

```
let add x y = x + y
```



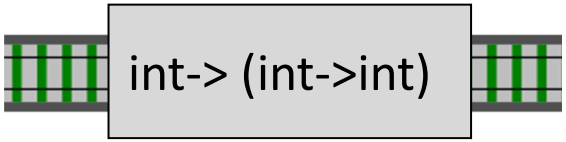
As a thing
(No parameters)

```
let add = (fun x y -> x + y)
```



One parameter

```
let add x = (fun y -> x + y)
```



```
let three = 1 + 2
```

"+" is a two parameter function

```
let three = (+) 1 2
```

```
let three = ((+) 1) 2
```

```
let three = 1 + 2
```

```
let three = (+) 1 2
```

```
let three = ((+) 1) 2
```

Missing a parameter?

```
let add1 = (+) 1
```

No, "+" is a one param function!

```
let three = add1 2
```

Pattern:
Partial application

Two parameters

```
let name = "Scott"
printfn "Hello, my name is %s" name
```

```
let name = "Scott"
(printfn "Hello, my name is %s") name
```

```
let name = "Scott"
let hello = (printfn "Hello, my name is %s")
hello name
```

One parameter

Can reuse "hello" in many places now!

Pattern:

Use partial application when working with lists

Partial application

```
let hello = printfn "Hello, my name is %s"
```

```
let names = ["Alice"; "Bob"; "Scott"]

names |> List.iter hello
```

Partial application

```
let add1 = (+) 1
let equals2 = (=) 2
```

```
[1..100]
|> List.map add1
|> List.filter equals2
```

Pattern:
Use partial application to do dependency injection

Persistence ignorant

```
type GetCustomer = CustomerId -> Customer
```

```
let getCustomerFromDatabase connection
                            (customerId:CustomerId) =
    // from connection
    // select customer
    // where customerId = customerId
```

This function requires a connection ☹

```
type of getCustomerFromDatabase =
                DbConnection -> CustomerId -> Customer
```

```
let getCustomer1 = getCustomerFromDatabase myConnection

// getCustomer1 : CustomerId -> Customer
```

The partially applied function does NOT require a connection – it's baked in! ☺

```
type GetCustomer = CustomerId -> Customer
```

```
let getCustomerFromMemory dict (customerId:CustomerId) =
    dict.Get(customerId)
```

This function requires a dictionary ☹

```
type of getCustomerFromMemory =
           Dictionary<Id,Customer> -> CustomerId -> Customer
```

```
let getCustomer2 = getCustomerFromMemory dict

// getCustomer2 : CustomerId -> Customer
```

The partially applied function does NOT require a dictionary – it's baked in! ☺

Pattern:
The Hollywood principle*: continuations

*Don't call us, we'll call you

Continuations

```
int Divide(int top, int bottom)
{
   if (bottom == 0)
   {
      throw new InvalidOperationException("div by 0");
   }
   else
   {
      return top/bottom;
   }
}
```

Method has decided to throw an exception

(who put it in charge?)

Continuations

Let the caller decide what happens

```
void Divide(int top, int bottom,
            Action ifZero, Action<int> ifSuccess)
{
   if (bottom == 0)
   {
      ifZero();
   }
   else
   {
      ifSuccess( top/bottom );
   }
}
```

what happens next?

Continuations

F# version

```
let divide ifZero ifSuccess top bottom =
   if (bottom=0)
   then ifZero()
   else ifSuccess (top/bottom)
```

Four parameters is a lot though!

Wouldn't it be nice if we could somehow "bake in" the two behaviour functions....

Continuations

```
let divide ifZero ifSuccess top bottom =
   if (bottom=0)
   then ifZero()
   else ifSuccess (top/bottom)
```

setup the functions to:
print a message

```
let ifZero1 () = printfn "bad"
let ifSuccess1 x = printfn "good %i" x


let divide1  = divide ifZero1 ifSuccess1

//test
let good1 = divide1 6 3
let bad1 = divide1 6 0
```

Partially apply the continuations

Use it like a normal function – only two parameters

Continuations

```
let divide ifZero ifSuccess top bottom =
   if (bottom=0)
   then ifZero()
   else ifSuccess (top/bottom)
```

setup the functions to:
return an Option

```
let ifZero2() = None
let ifSuccess2 x = Some x


let divide2  = divide ifZero2 ifSuccess2

//test
let good2 = divide2 6 3
let bad2 = divide2 6 0
```

Partially apply the continuations

Use it like a normal function –
only two parameters

Continuations

```
let divide ifZero ifSuccess top bottom =
   if (bottom=0)
   then ifZero()
   else ifSuccess (top/bottom)
```

setup the functions to:
throw an exception

```
let ifZero3() = failwith "div by 0"
let ifSuccess3 x = x


let divide3  = divide ifZero3 ifSuccess3

//test
let good3 = divide3 6 3
let bad3 = divide3 6 0
```

Partially apply the continuations

Use it like a normal function –
only two parameters

Pattern:
Chaining callbacks with continuations

Pyramid of doom: null testing example

```
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...

Pyramid of doom: async example

```
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

Pyramid of doom: null example

```
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!

Pyramid of doom: option example

```
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...

Pyramid of doom: option example

```
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
```

Pyramid of doom: option example

```
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
```

Pyramid of doom: option example

```
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
```

Crying out to be parameterized!

Parameterize all the things!

```
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
```

Much cleaner code now

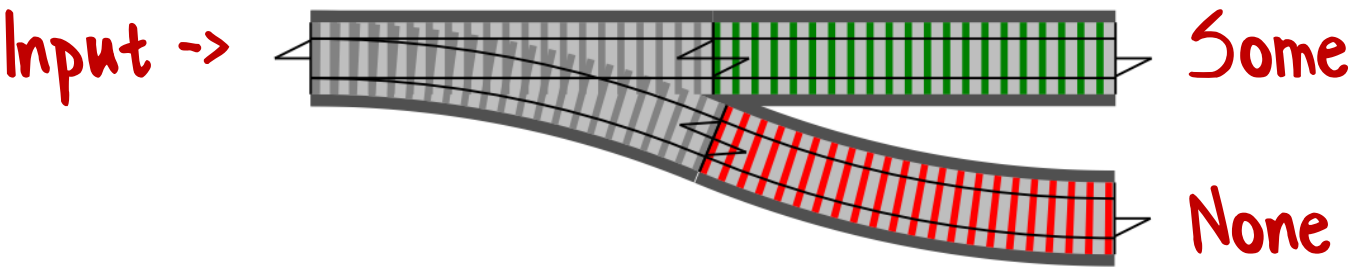
```
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo (fun z -> Some z)
```

MONADS

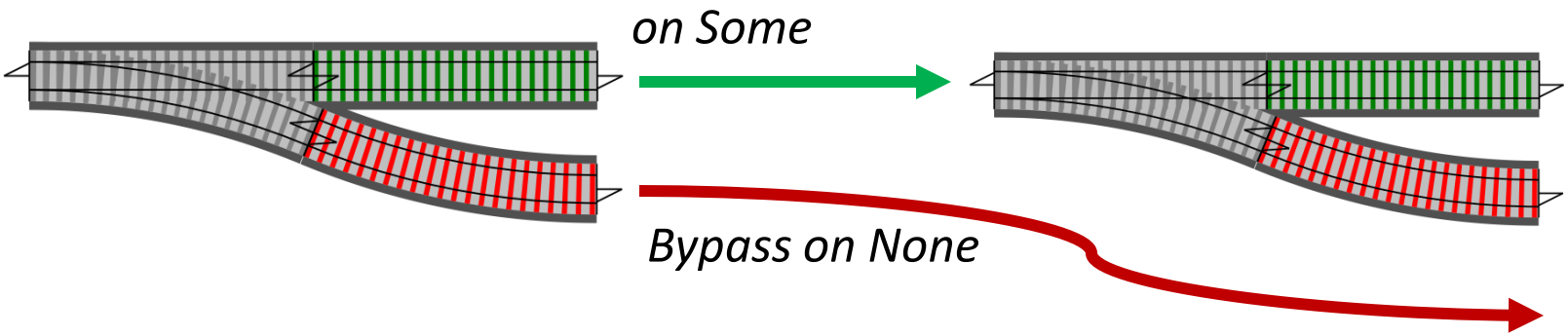
a.k.a. chaining continuations

(ok, it's a bit more complicated)

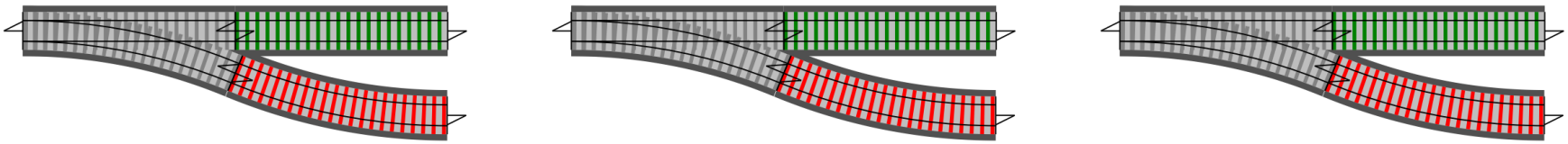
A switch analogy



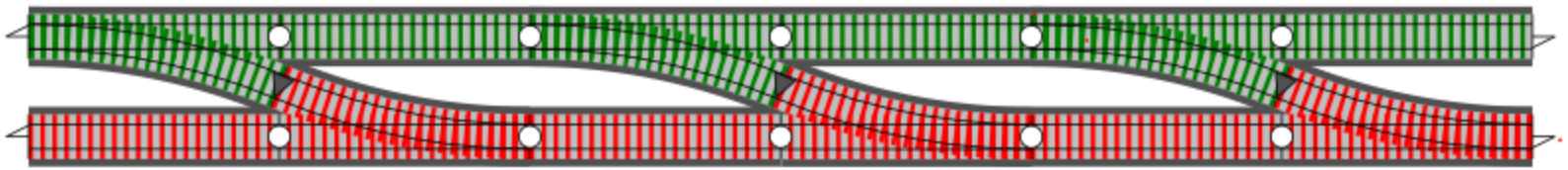
Connecting switches



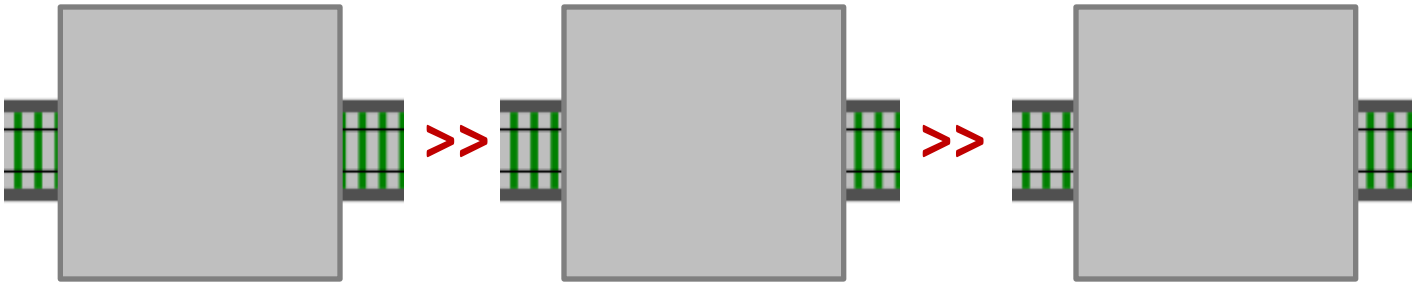
Connecting switches



Connecting switches

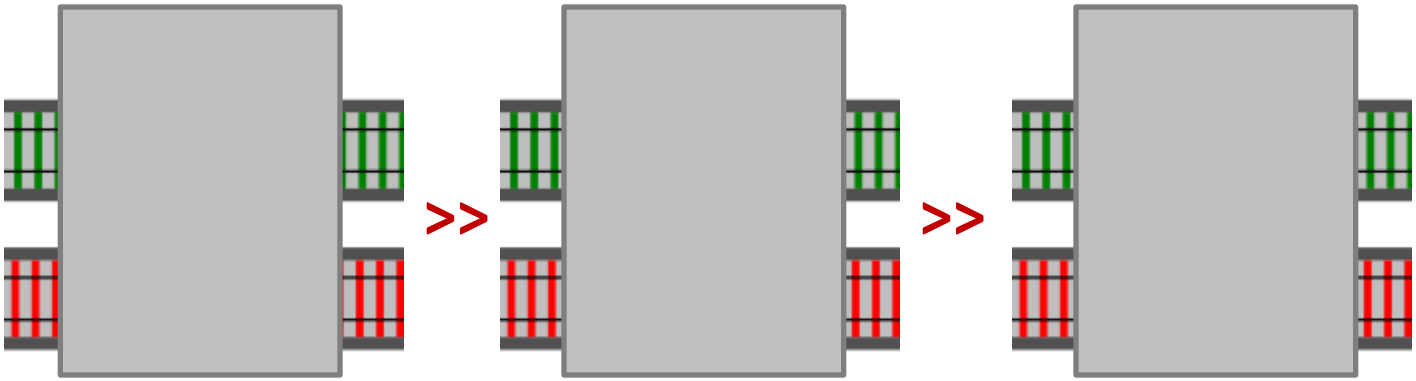


Composing switches



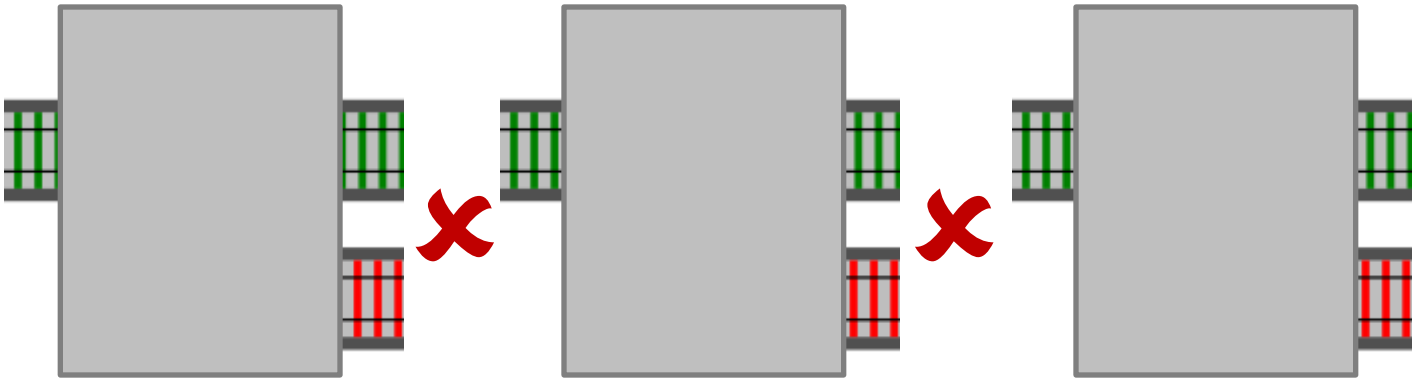
Composing one-track functions is fine...

Composing switches



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

Composing switches



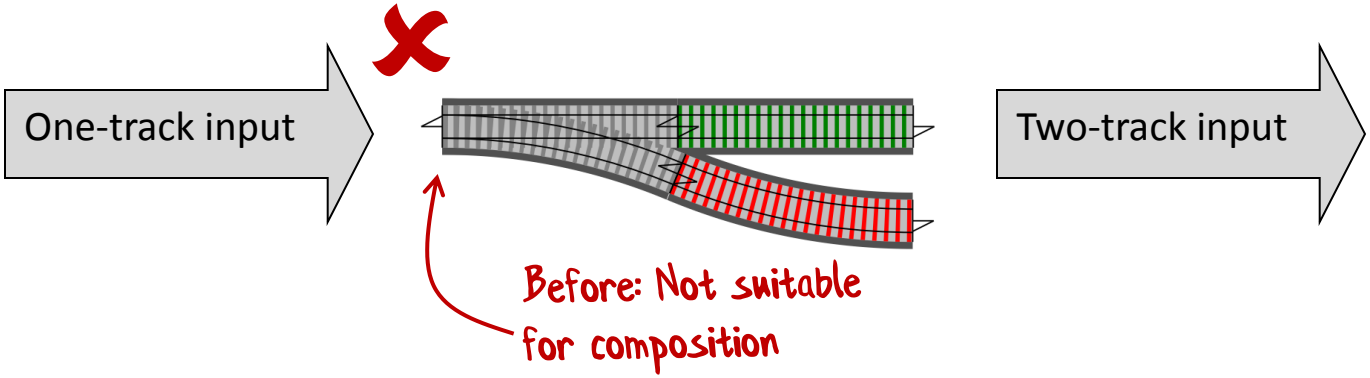
... but composing switches is not allowed!

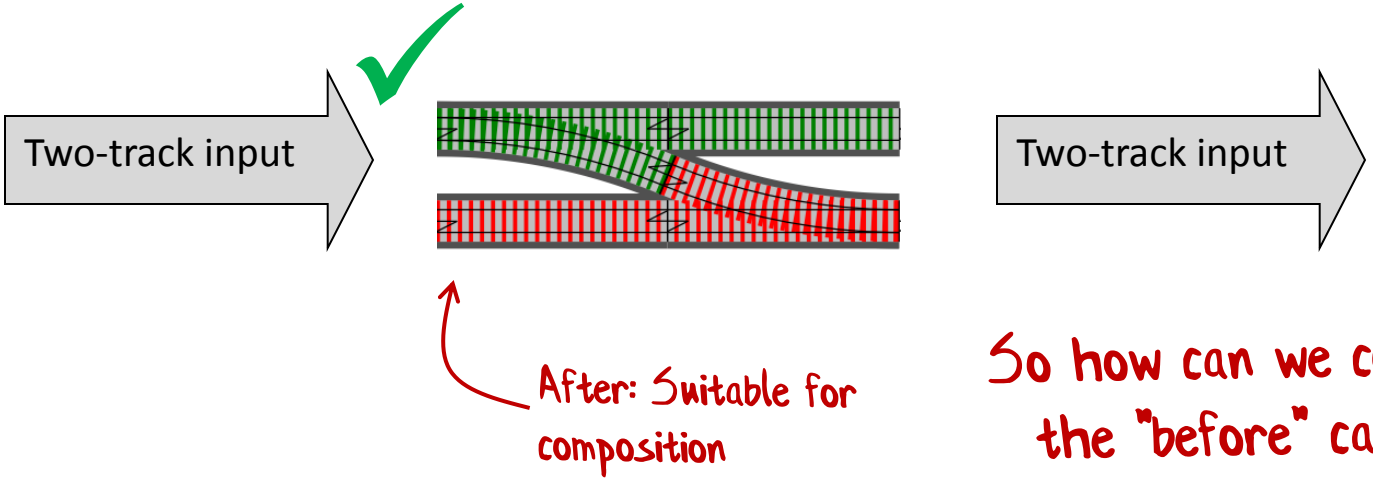
How to combine the mismatched functions?

“Bind” is the answer!
Bind all the things!

FP’ers get excited by bind

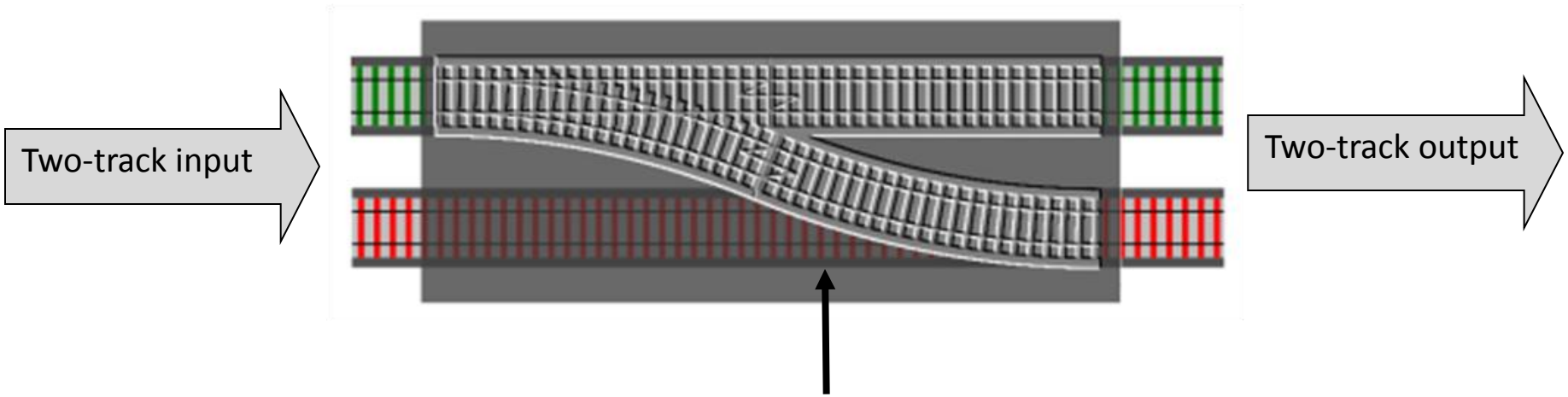
Composing switches





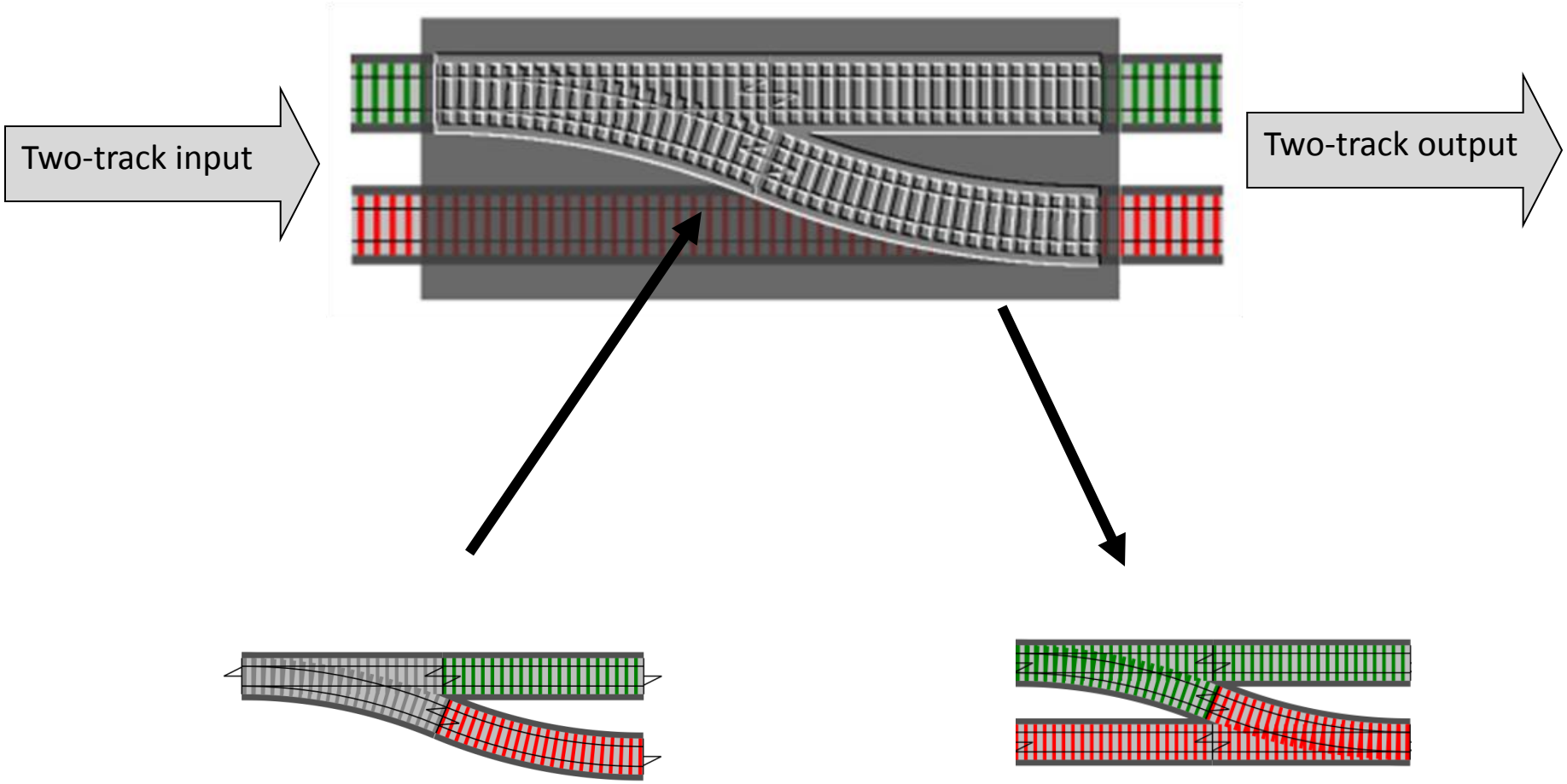
So how can we convert from the "before" case to the "after" case?

Building an adapter block

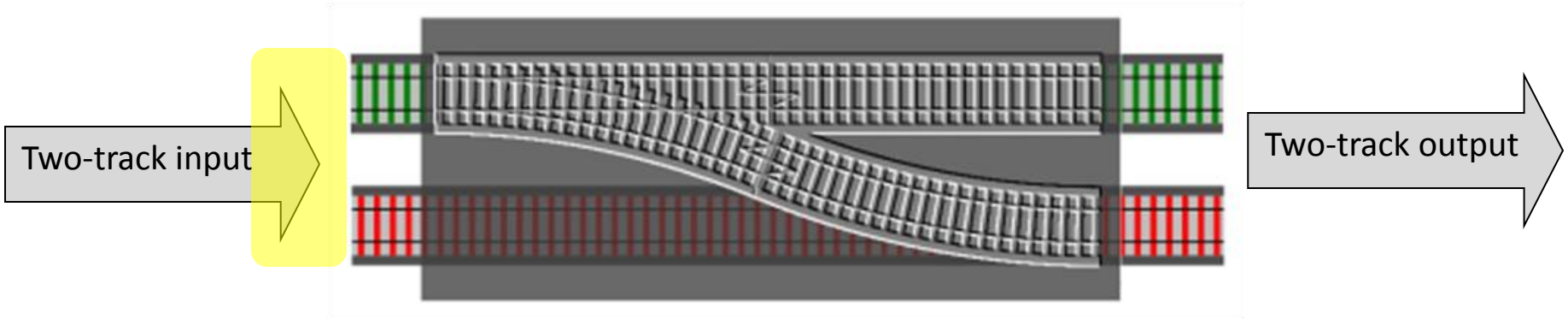


Slot for switch function

Building an adapter block

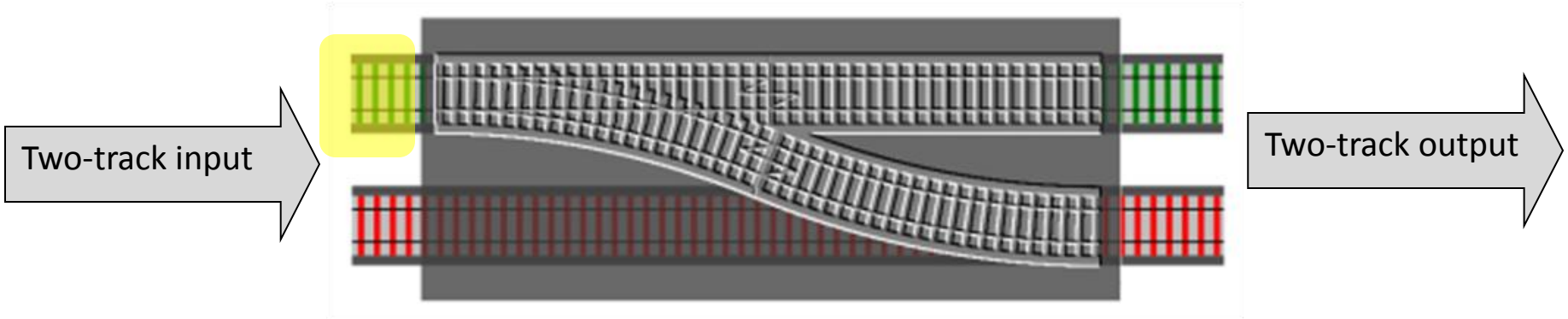


Building an adapter block



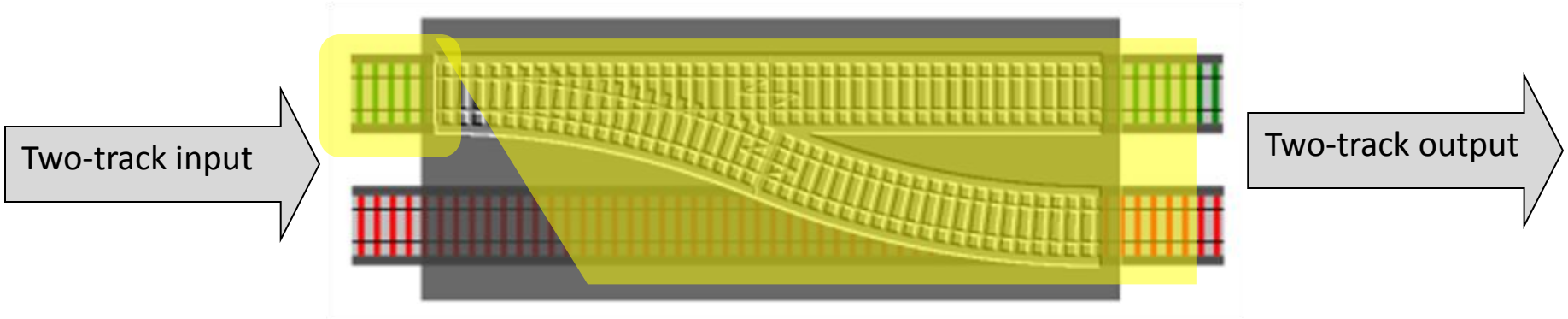

```
let bind nextFunction optionInput =
   match optionInput with
   | Some s -> nextFunction s
   | None -> None
```

Building an adapter block




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

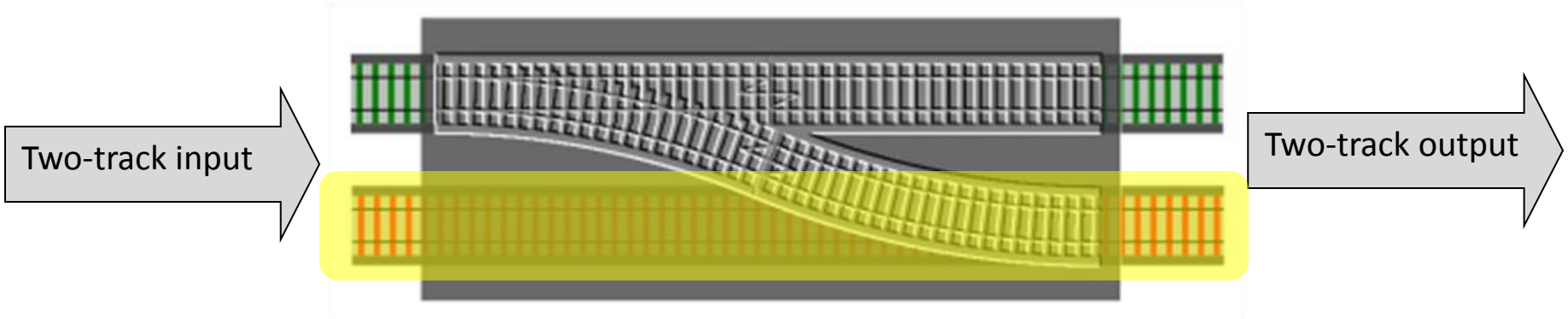
Building an adapter block




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

This is a continuation

Building an adapter block




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

Pattern:

Use bind to chain options

Pyramid of doom: using bind

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

```
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
```

Pyramid of doom: using bind

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

```
let example input =
    doSomething input
    |> bind doSomethingElse
    |> bind doAThirdThing
    |> bind (fun z -> Some z)
```

No pyramids!

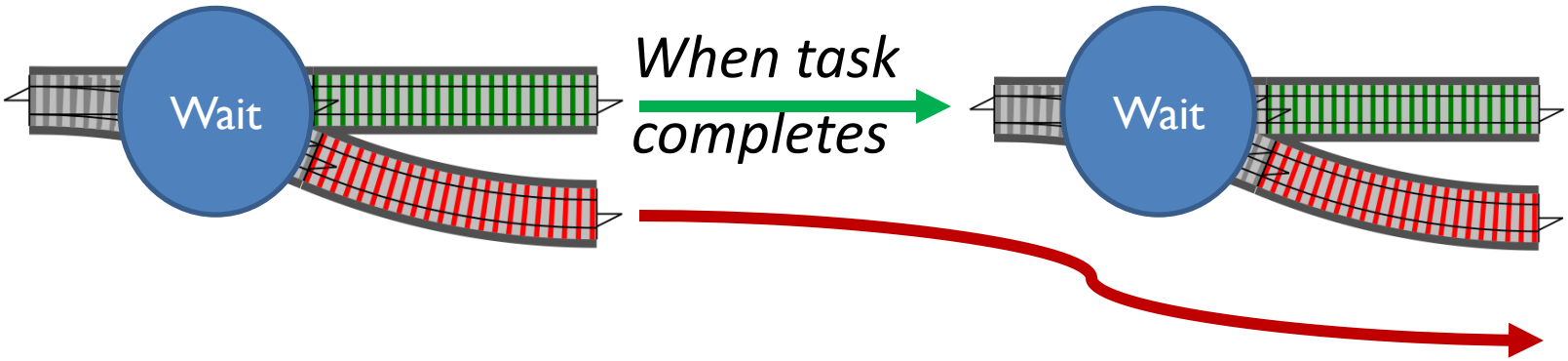
Code is linear and clear.

This pattern is called "monadic bind"

Pattern:

Use bind to chain tasks

Connecting tasks



Pyramid of doom: using bind for tasks

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

a.k.a "promise" "future"

```
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
```

Pyramid of doom: using bind for tasks

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

```
let taskExample input =
    startTask input
    |> taskBind startAnotherTask
    |> taskBind startThirdTask
    |> taskBind (fun z -> z)
```

This pattern is also a "monadic bind"

Pattern:

Use bind to chain error handlers

Use case without error handling

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

  return "OK";
}
```

Use case with error handling

```
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";
}
```

Use case with error handling

```
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";
}
```

Use case with error handling

```
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";
}
```

Use case with error handling

```
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";
}
```

Use case with error handling

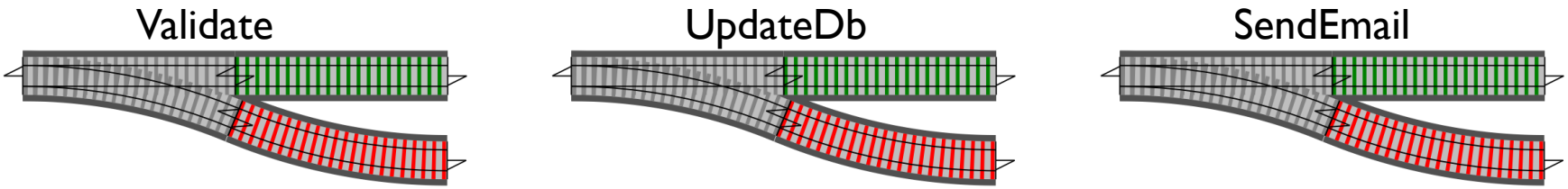
```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  try {
    var result = db.updateDbFromRequest(request);
    if (!result) {
     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

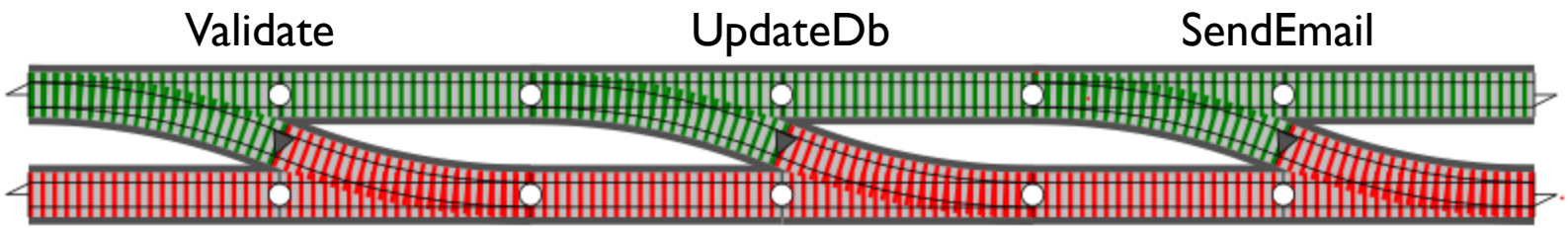
A structure for managing errors


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

Connecting switches



Connecting switches



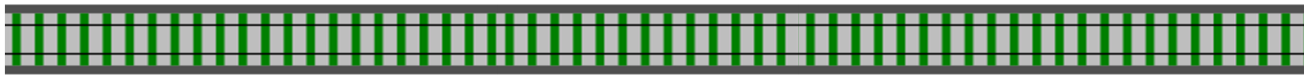
This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

Functional flow without error handling

Before

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

One track

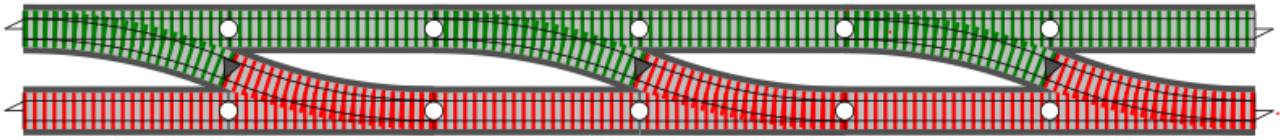


Functional flow with error handling

After

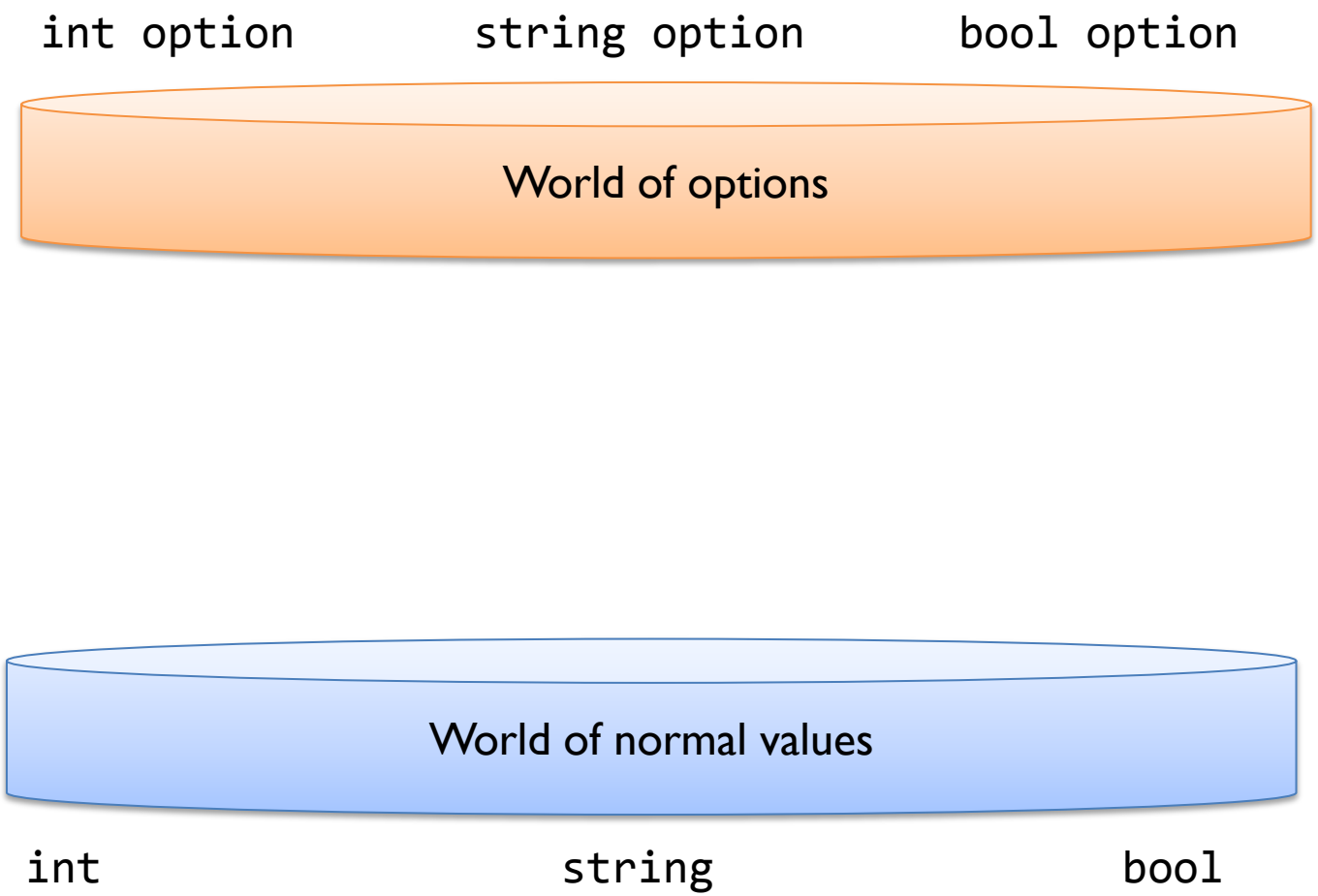
```
let updateCustomerWithErrorHandling =
  receiveRequest
  |> validateRequest
  |> canonicalizeEmail
  |> updateDbFromRequest
  |> sendEmail
  |> returnMessage
```

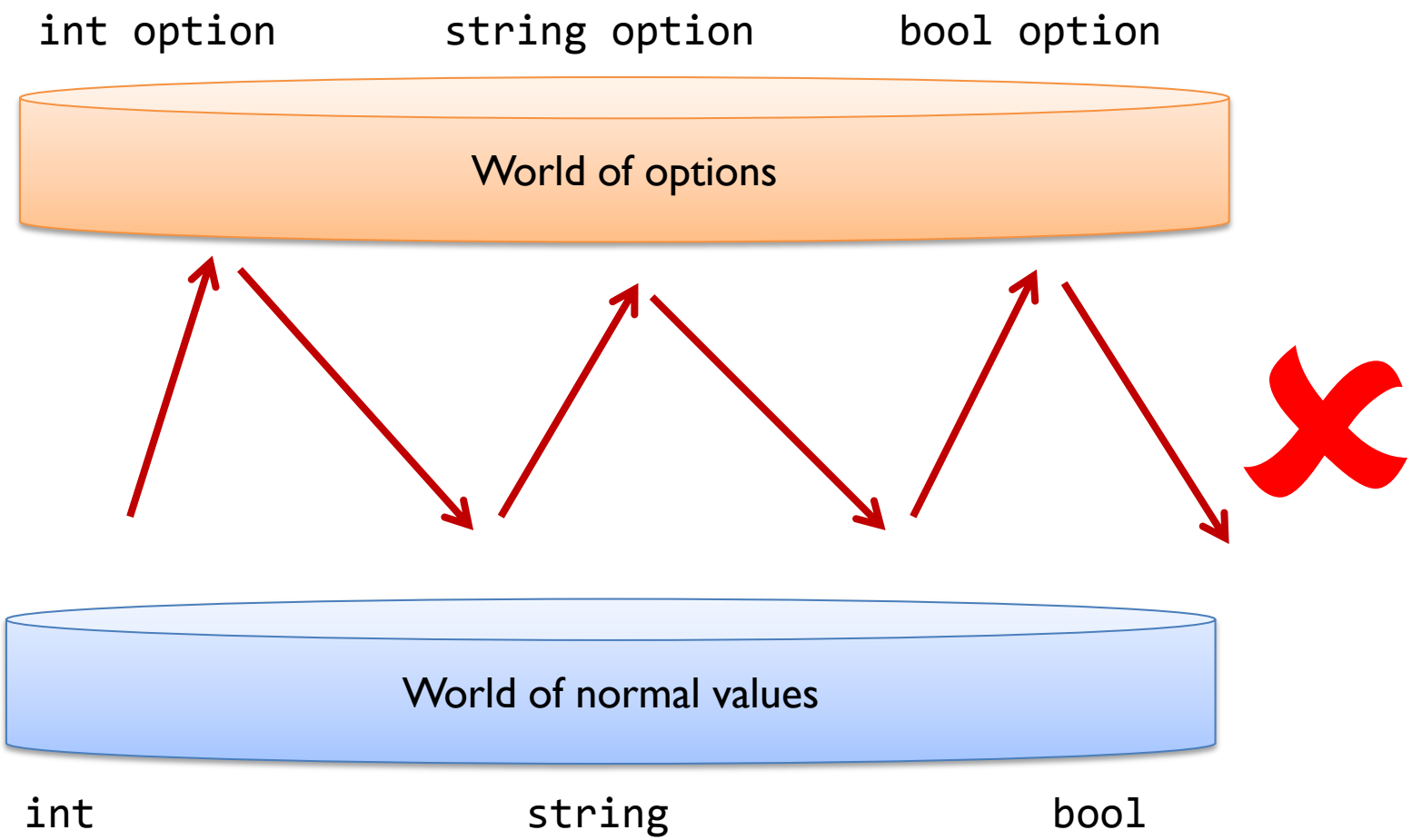
Two track

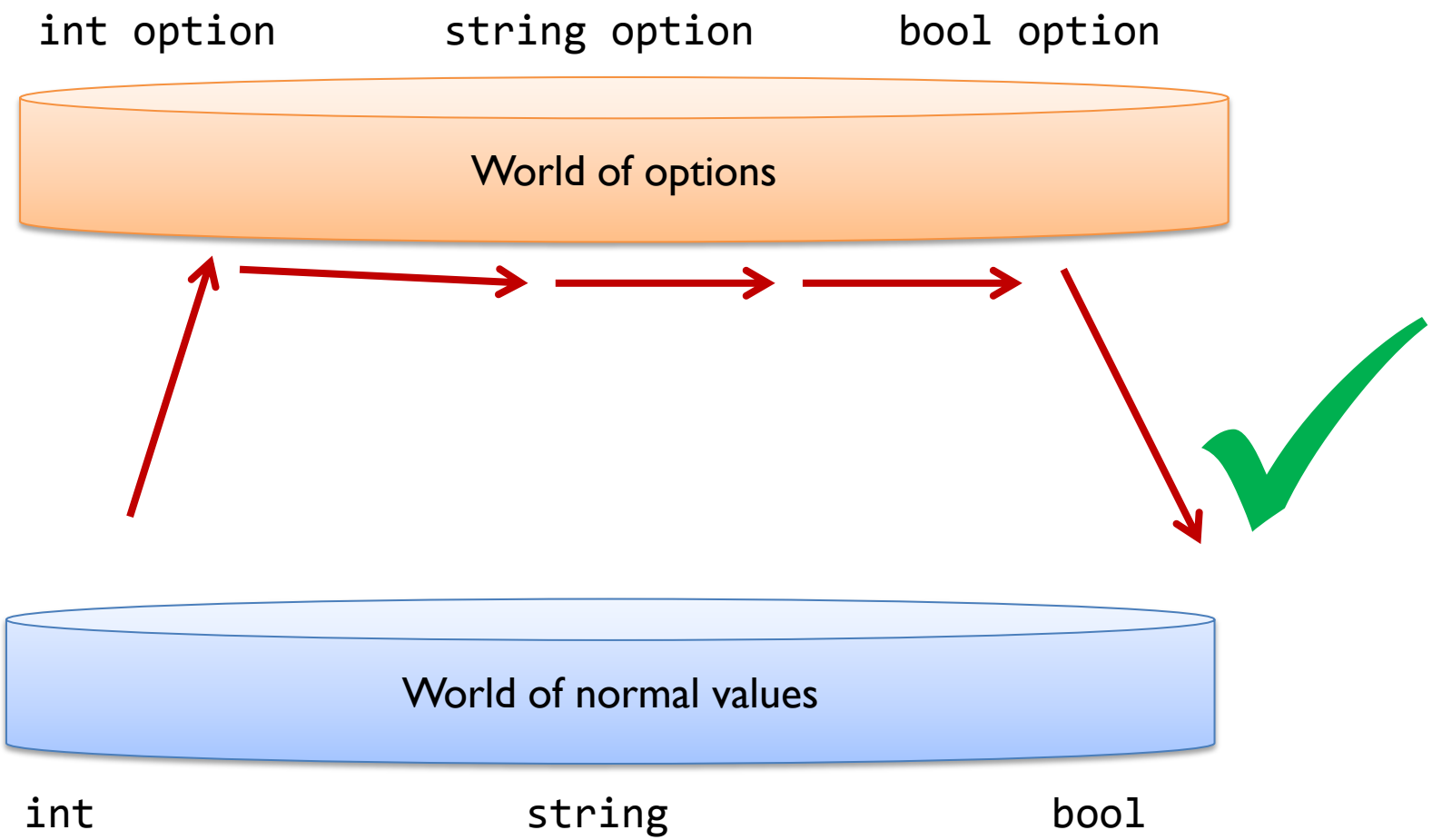


See my talk on Friday or
fsharpforfunandprofit.com/rop

MAPS


int option
string option
bool option
World of options
World of normal values
int
string
bool


int option
string option
bool option
World of options
World of normal values
int
string
bool


int option
string option
bool option
World of options
World of normal values
int
string
bool

How not to code with options

Let's say you have an int wrapped in an Option, and you want to add 42 to it:

```
let add42 x = x + 42
```

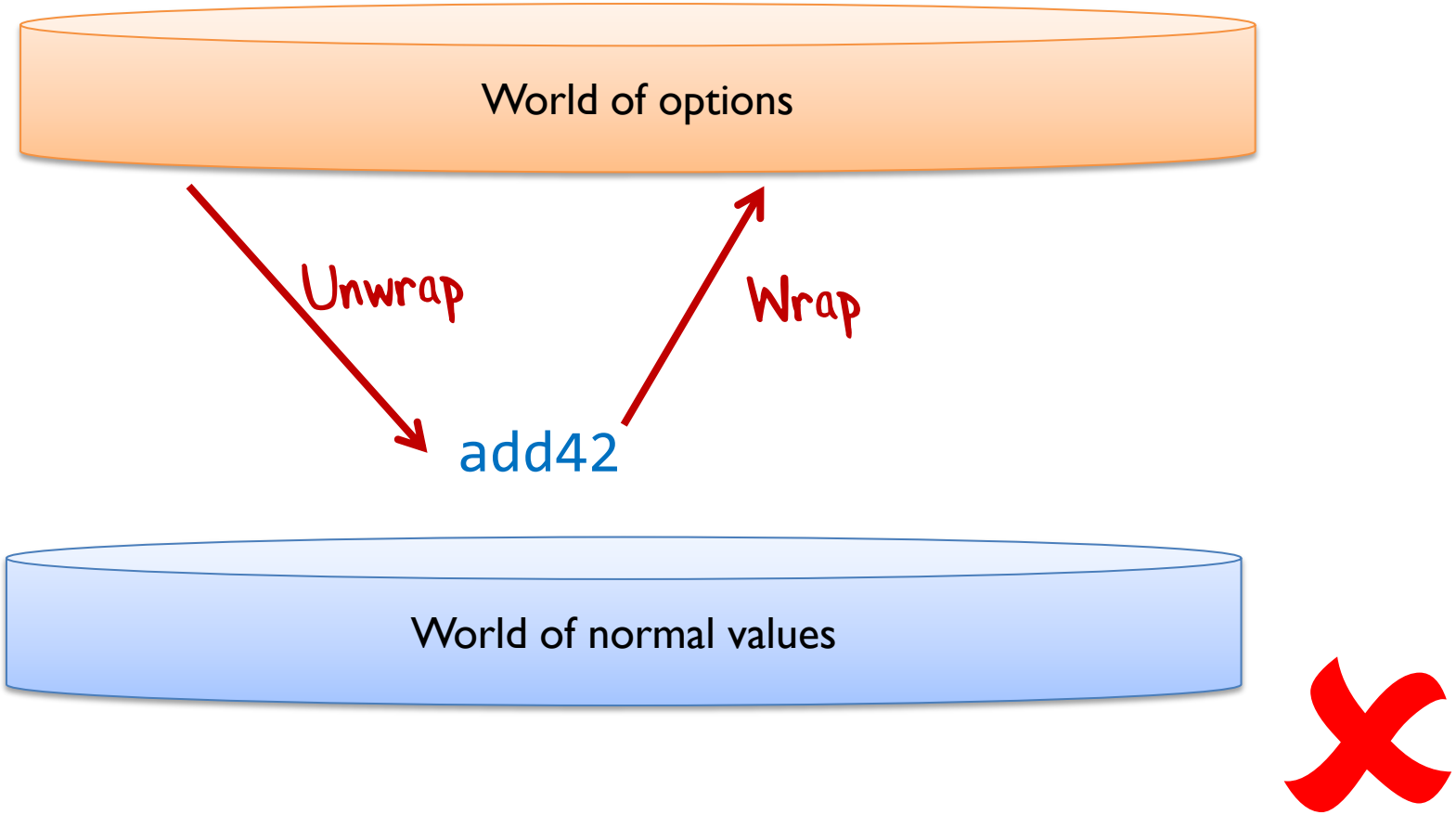
Works on normal values

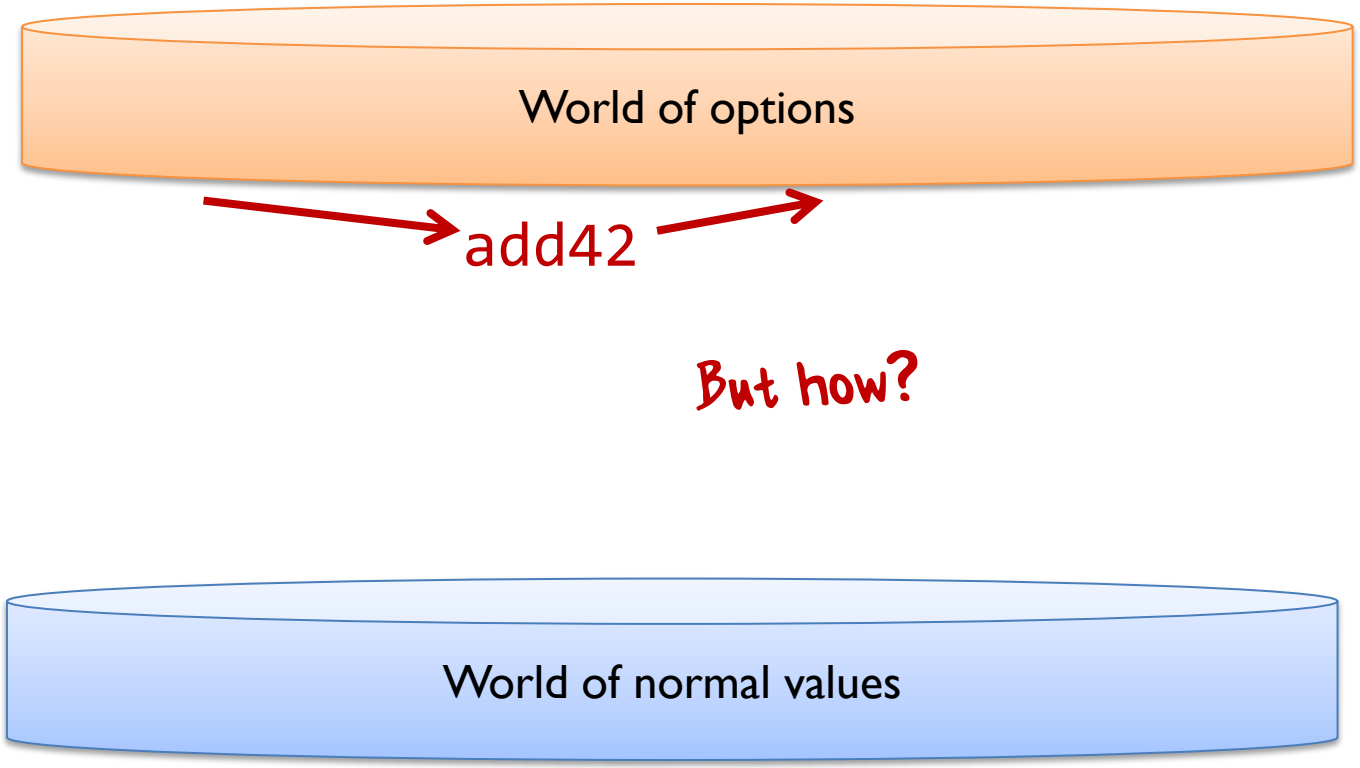
```
let add42ToOption opt =
    if opt.IsSome then
        let newVal = add42 opt.Value
        Some newVal
    else
        None
```

Unwrap

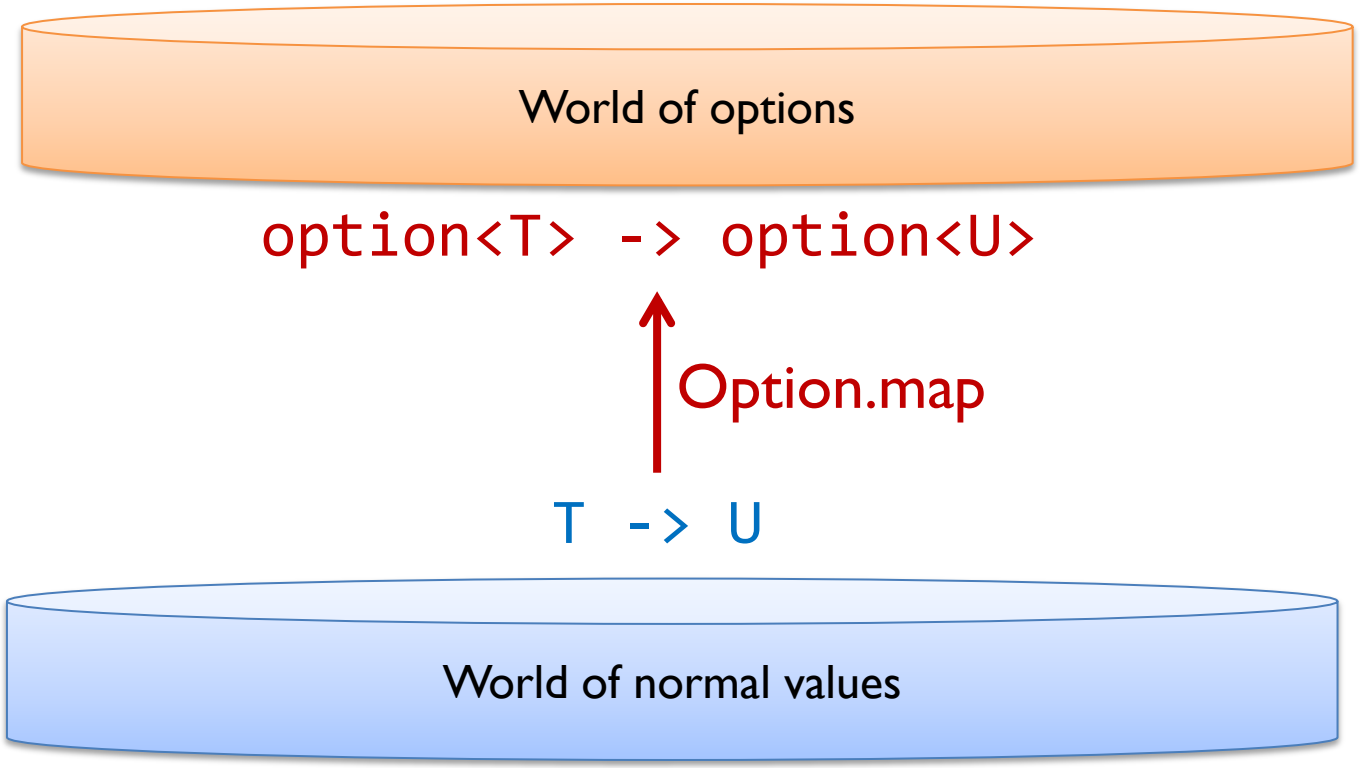
Apply

Wrap again


World of options
Unwrap
Wrap
add42
World of normal values


World of options
add42
But how?
World of normal values

Lifting


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

World of options

```
Some 1 |>  add42ToOption      // Some 43
```

↑ Map

```
1 |>       add42             // 43
```

World of normal values

The right way to code with options

Let's say you have an int wrapped in an Option, and you want to add 42 to it:

```
let add42 x = x + 42
1 |> add42                          // 43
```

```
let add42ToOption = Option.map add42
Some 1 |> add42ToOption             // Some 43
```

The right way to code with options

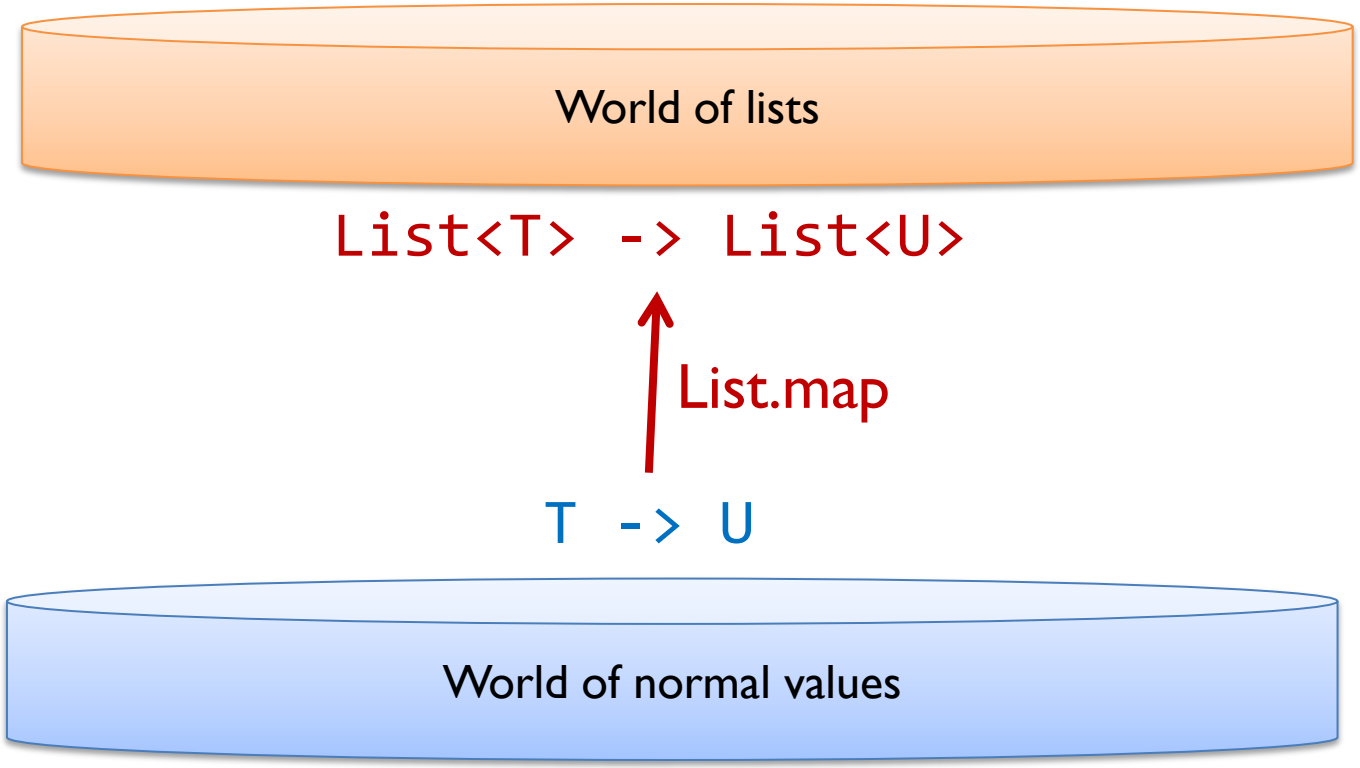
Let's say you have an int wrapped in an Option, and you want to add 42 to it:

```
let add42 x = x + 42
1 |> add42                          // 43
```

```
Some 1 |> Option.map add42  // Some 43
```

Often no need to bother creating a temp function

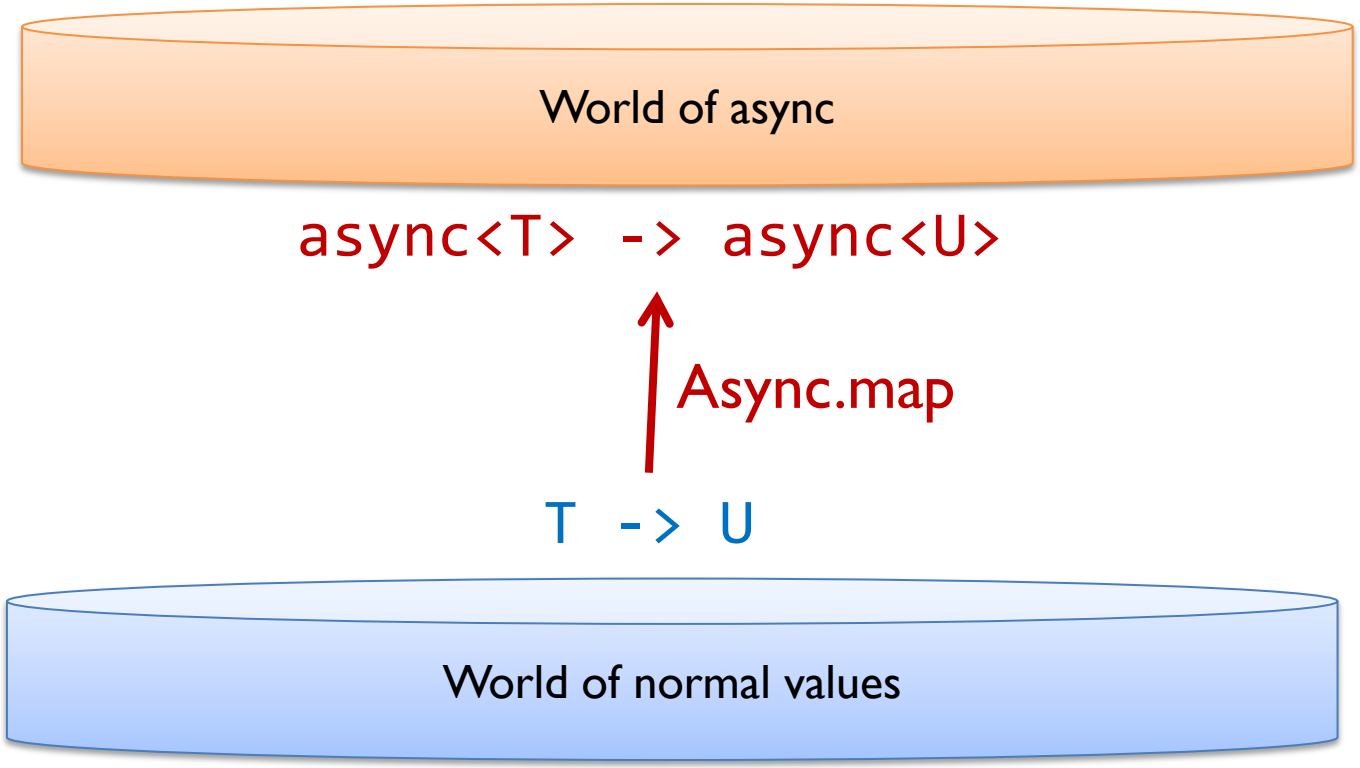
Lifting to lists



The right way to code with wrapped types

```
[1;2;3] |> List.map add42

            // [43;44;45]
```

Lifting to async



Guideline:

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

Guideline:

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

Terminology alert:
Mappable types are "functors"

MONOIDS


WARNING
Mathematics
Ahead

Thinking like a mathematician

$$1 + 2 = 3$$

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

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

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

The generalization

- 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!

- **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.

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

- **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.

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

- **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.

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

- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Benefit**: Divide and conquer, parallelization, and incremental accumulation.

$$1 + 2 + 3 + 4$$

- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Benefit**: Divide and conquer, parallelization, and incremental accumulation.

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

$$3 + 7$$

- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Benefit**: Divide and conquer, parallelization, and incremental accumulation.

$$(1 + 2 + 3)$$

- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Benefit**: Divide and conquer, parallelization, and incremental accumulation.

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

- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Benefit**: Divide and conquer, parallelization, and incremental accumulation.

(6) + 4

Issues with reduce

- 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?

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

If zero is missing, it is called a semigroup

Pattern:
Simplifying 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} ]
```

Any combination of monoids is also a monoid

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

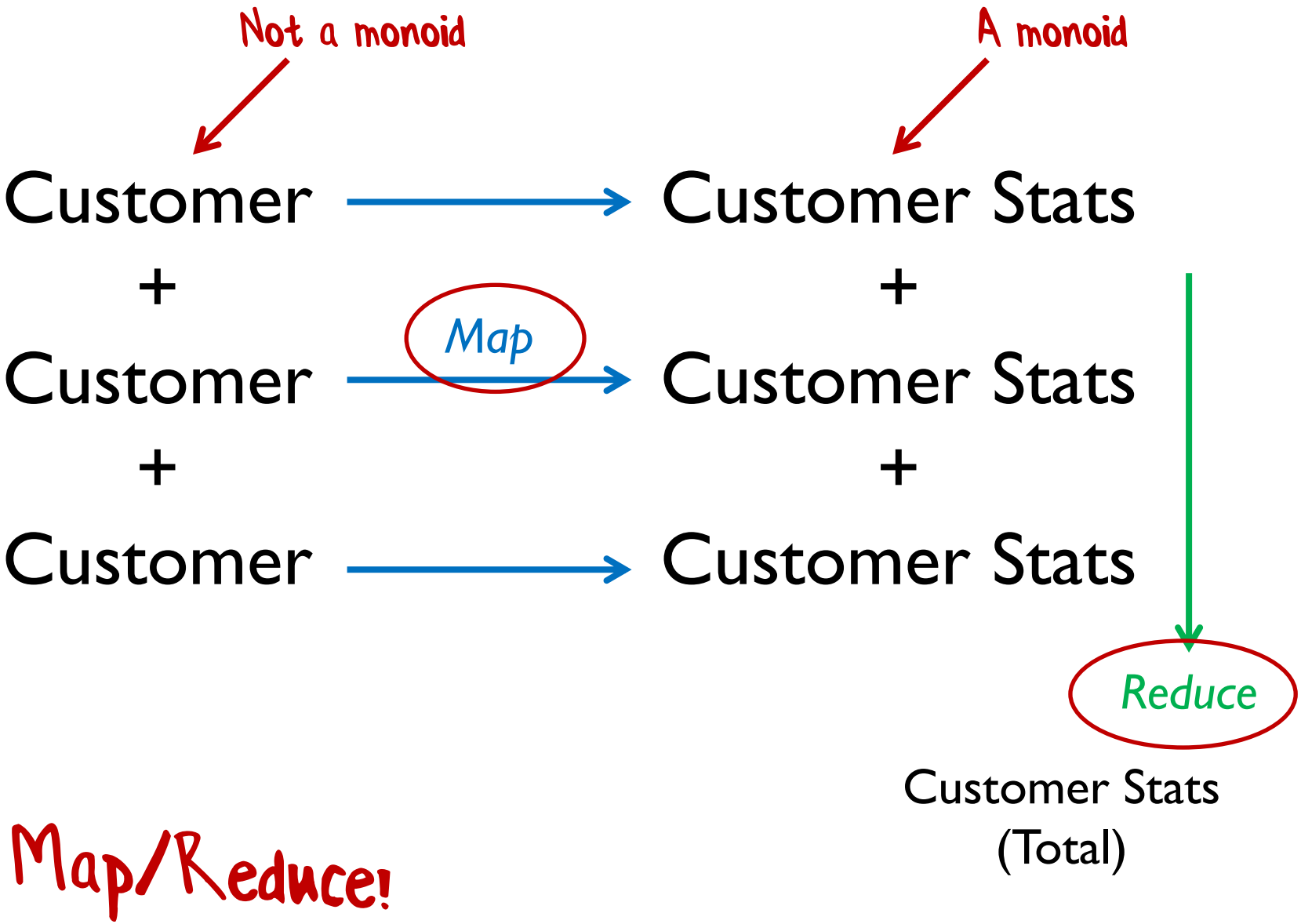
Write a pairwise combiner

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

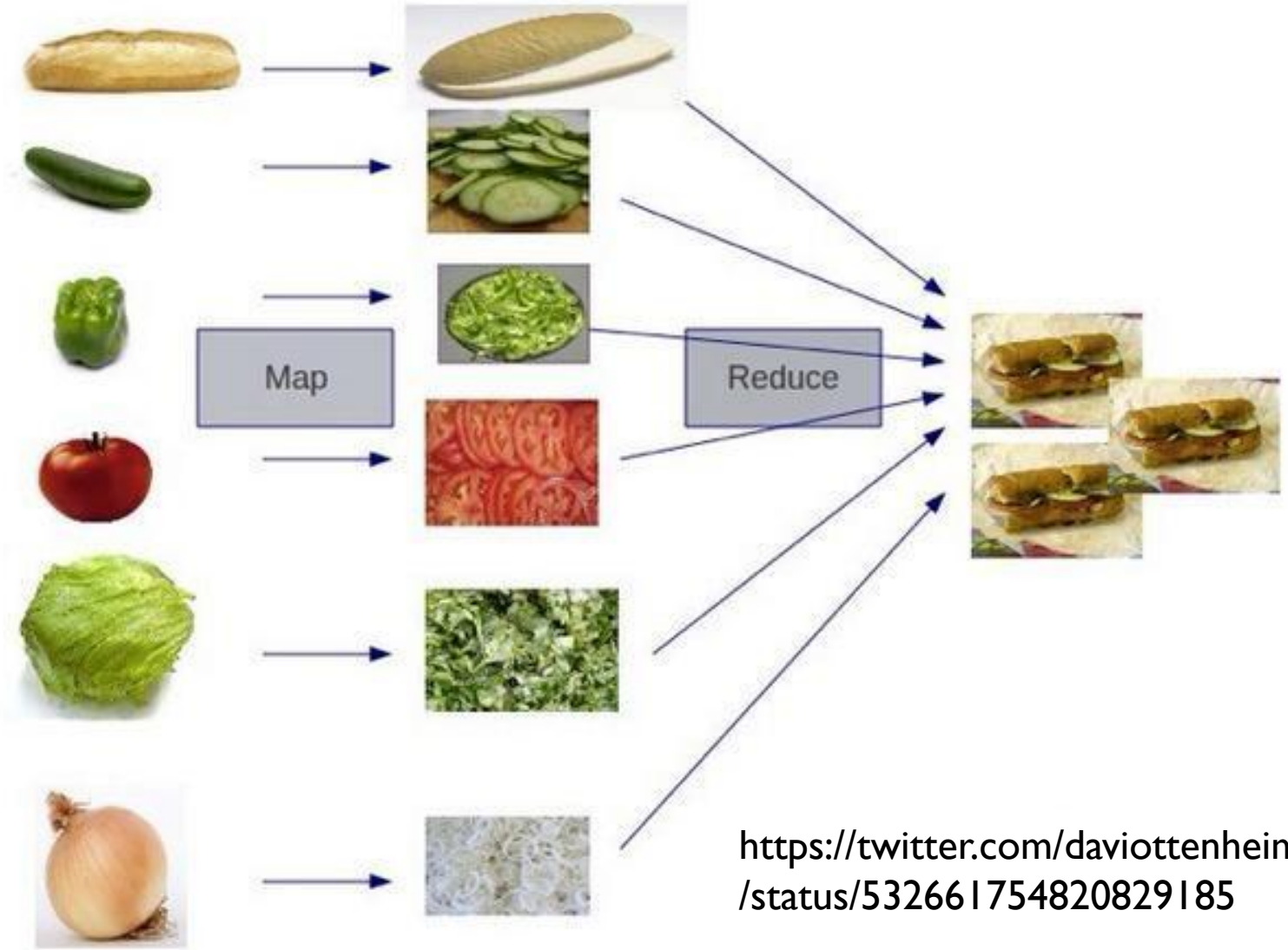
Profit!

Pattern:

Convert non-monoids to monoids


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

Hadoop make me a sandwich



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

Guideline:
Convert expensive monoids
to cheap monoids

Strings are monoids

A monoid

Log file (Mon) → Summary (Mon)

+ +

Log File (Tue) —*Map*→ Summary (Tue)

+ +

Log File (Wed) → Summary (Wed)

=

Really big file

Much more efficient for incremental updates

"Monoid homomorphism"

Pattern:
Seeing monoids everywhere

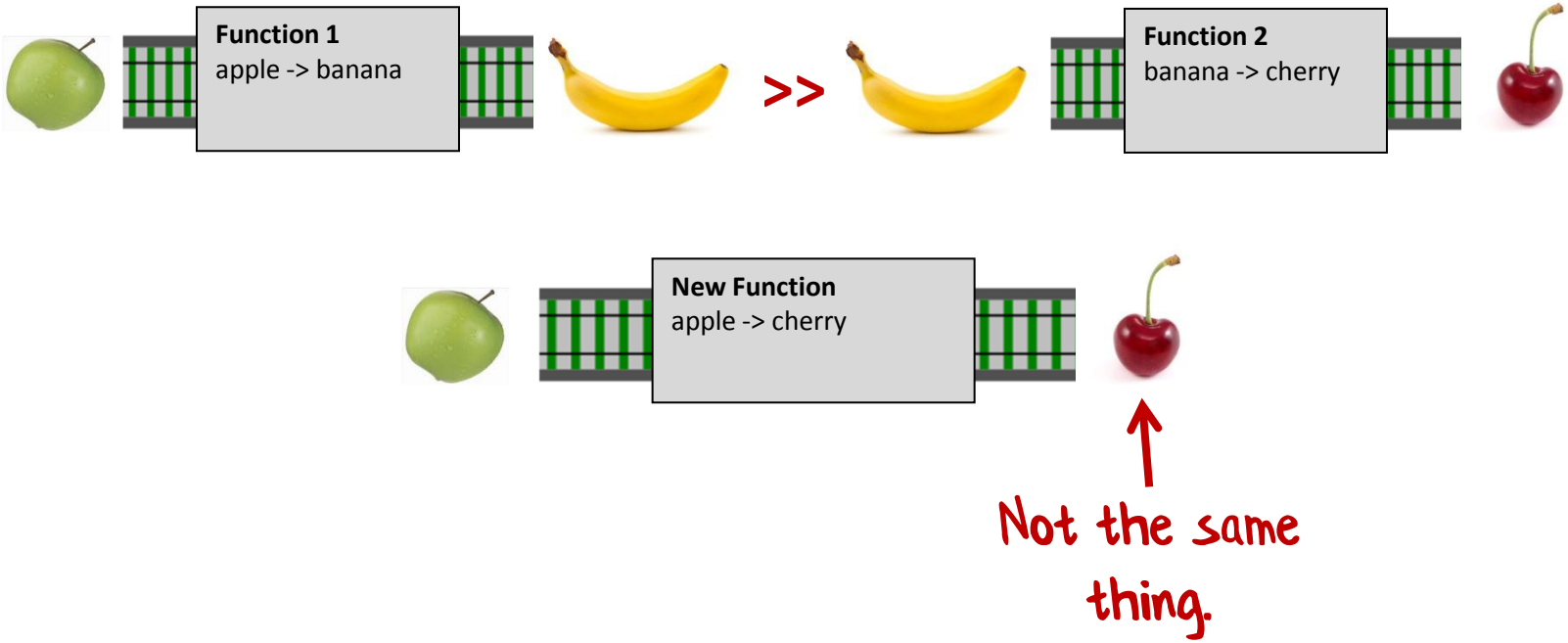
Monoids in the real world

Metrics guideline:
Use counters rather than rates

Alternative metrics guideline:
Make sure your metrics are monoids

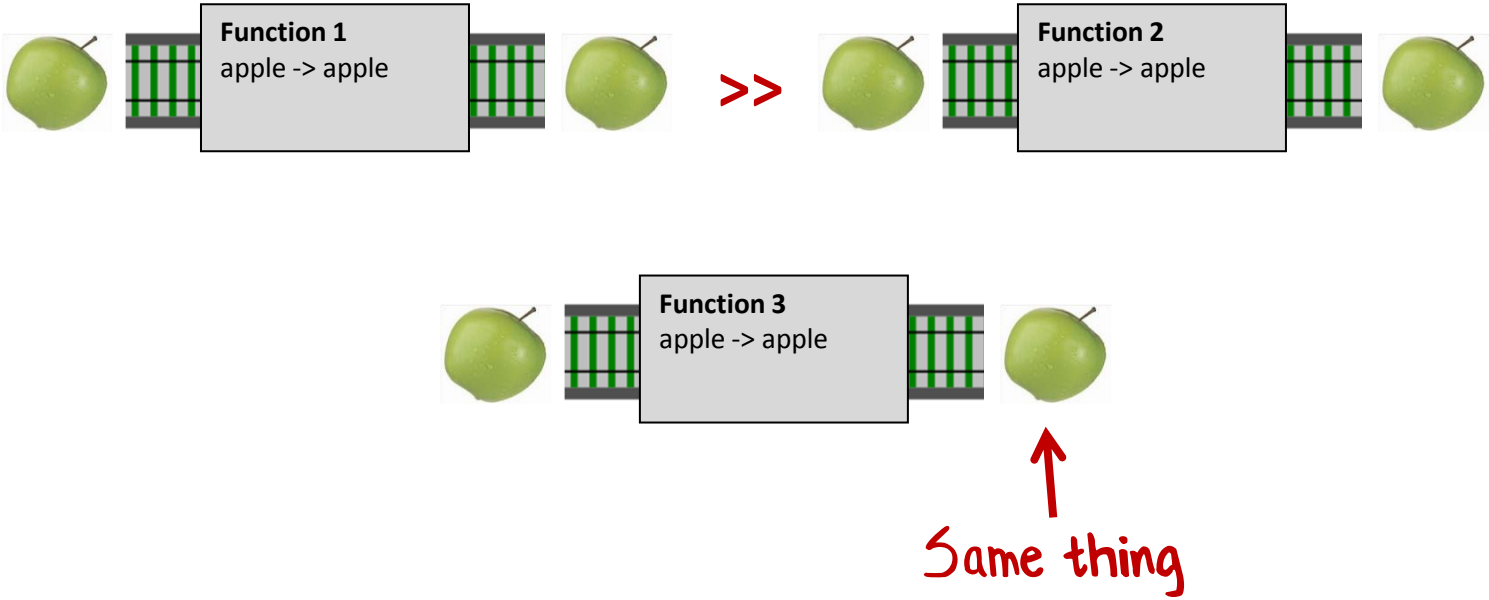
- incremental updates
- can handle missing data

Is function composition a monoid?



Not a monoid ☹

Is function composition a monoid?



A monoid! ☺

Is function composition a monoid?

Functions where the input and output are the *same* type are monoids! What shall we call these kinds of functions?

"Functions with same type of input and output"

Is function composition a monoid?

Functions where the input and output are the *same* type are monoids! What shall we call these kinds of functions?

~~"Functions with same type of input and output"~~

"Endomorphisms"

All endomorphisms are monoids

Endomorphisms

```
let plus1 x = x + 1            // int->int
```

```
let times2 x = x * 2           // int->int
```

```
let subtract42 x = x - 42      // int->int
```

Reduce

```
plus1ThenTimes2ThenSubtract42 // int->int
```

Another endomorphism!

Event sourcing

Event application function:

Is an endomorphism

Event -> State -> State

Any function containing an endomorphism can be converted into a monoid!

Partial application of event

Endomorphism

```
apply event1          // State -> State
```

```
apply event2          // State -> State
```

```
apply event3          // State -> State
```

Reduce

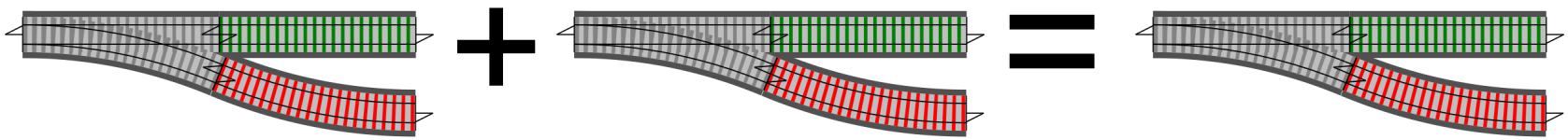
```
applyAllEventsAtOnce    // State -> State
```

Another endomorphism!

- incremental updates
- can handle missing events

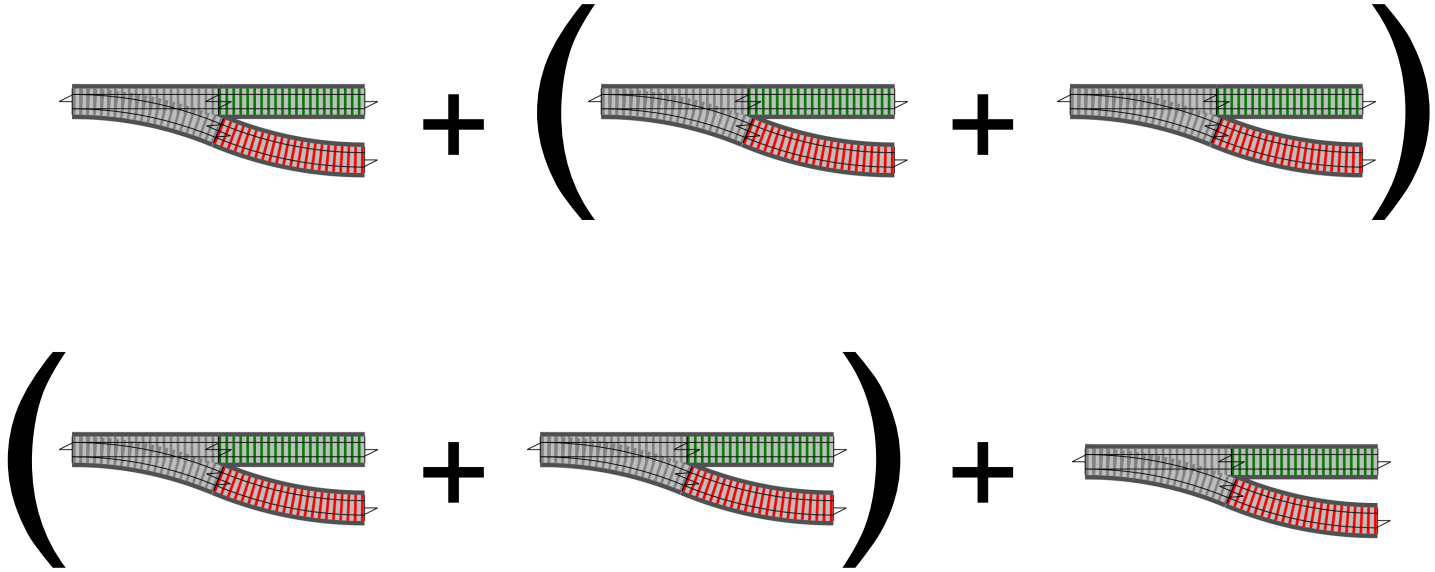
Monads vs. monoids?

Series combination



Result is same kind of thing (Closure)

Series combination

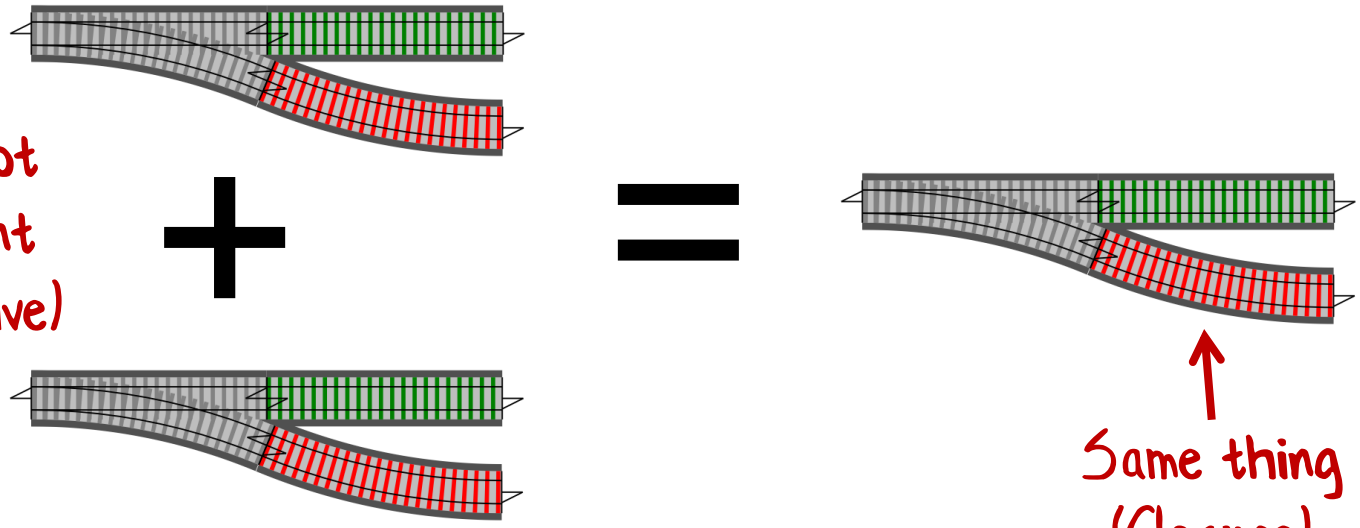


Order not important (Associative)

Monoid!

Parallel combination

Order not important (Associative)



Same thing (Closure)

Monoid!

Monad laws

- The Monad laws are just the monoid definitions in diguise
  - Closure, Associativity, Identity
- What happens if you break the monad laws?
  - You go to jail
  - You lose monoid benefits such as aggregation

A monad is just a monoid in the category of endofunctors!

A monad is just a monoid in the category of endofunctors!

THANKS!

More F#
here!